

Jayme Luiz  
Szwarcfiter

SÉRIE  
**SBC**  
SOCIEDADE  
BRASILEIRA DE  
COMPUTAÇÃO

# TEORIA COMPUTACIONAL DE GRAFOS

---

## Os algoritmos

Com programas Python por  
Fabiano S. Oliveira e  
Paulo E. D. Pinto

ELSEVIER

Material  
na WEB  
[www.evolution.com.br](http://www.evolution.com.br)

Jayme Luiz Szwarcfiter

TEORIA COMPUTACIONAL  
DE GRAFOS:  
OS ALGORITMOS

ELSEVIER



**Jayme Luiz  
Szwarcfiter**



# **TEORIA COMPUTACIONAL DE GRAFOS**

---

## **Os algoritmos**

**Com programas Python por  
Fabiano S. Oliveira e  
Paulo E. D. Pinto**

**ELSEVIER**



© 2018, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

ISBN: 978-85-352-8884-1

ISBN (versão digital): 978-85-352-8885-8

**Copidesque:** Augusto Coutinho

**Elsevier Editora Ltda.**

**Conhecimento sem Fronteiras**

Rua da Assembléia, nº 100 – 6º andar  
20011-904 – Centro – Rio de Janeiro – RJ

Rua Quintana, 753 – 8º andar  
04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800 026 53 40

atendimento1@elsevier.com

Consulte nosso catálogo completo, os últimos lançamentos e os serviços exclusivos no site [www.elsevier.com.br](http://www.elsevier.com.br).

**NOTA**

Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso serviço de Atendimento ao Cliente para que possamos esclarecer ou encaminhar a questão.

Para todos os efeitos legais, nem a editora, nem os autores, nem os editores, nem os tradutores, nem os revisores ou colaboradores, assumem qualquer responsabilidade por qualquer efeito danoso e/ou malefício a pessoas ou propriedades envolvendo responsabilidade, negligência etc. de produtos, ou advindos de qualquer uso ou emprego de quaisquer métodos, produtos, instruções ou ideias contidos no material aqui publicado.

A Editora

**CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO  
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ**

---

S998t

Szwarcfiter, Jayme Luiz

Teoria Computacional de Grafos / Jayme Luiz Szwarcfiter. – 1. ed. – Rio de Janeiro : Elsevier, 2018.  
ii.

Inclui índice

ISBN 978-85-352-8884-1

1. Ciéncia da computaçáo. 2. Programação (Computadores). 3. Algorítmicos. 4. Estruturas de dados (Computação). I. Título.

---

18-47019

CDD:005.1

CDU:004.41

---



*À Cristina, Cláudio, Lila e Ana*

Página deixada intencionalmente em branco

---

# Prefácio

---

Quando recebi o aguardado novo livro do Jayme sobre Teoria dos Grafos, logo lembrei das palavras do seu colega na UFRJ, o grande cientista Carlos Chagas Filho: "Na Universidade se ensina porque se pesquisa."

O seu livro anterior, o pioneiro *Grafos e Algoritmos Computacionais* lançado 35 anos atrás, é um clássico e está esgotado há mais de 20 anos. Ele nasceu na UFRJ como texto de cursos na graduação e na pós-graduação, já que na época Jayme lecionava Algoritmos e Grafos na graduação do Instituto de Matemática e Teoria Computacional de Grafos na pós-graduação da COPPE. A primeira versão do texto foi apresentada no início dos anos 1980, por ocasião da Terceira Escola de Computação, e foi simultânea às primeiras teses de doutorado orientadas pelo Jayme na COPPE.

Como o anterior, este novo livro marca uma trajetória de ensino e pesquisa: as 40 teses de doutorado orientadas pelo Jayme comprovam o efeito multiplicador do texto, o sucesso do convite que muitos aceitaram através da leitura de um texto introdutório em Teoria dos Grafos.

O perfil original permanece: o seu caráter introdutório é um convite para os que querem ser apresentados ao tema, o seu caráter matemático e computacional amplia o alcance do texto para os cursos de engenharia, e o seu caráter de pesquisa fica evidente pelas detalhadas notas bibliográficas que concluem cada capítulo.

Muitas novidades foram incluídas, desde o título: todos os capítulos trazem agora uma nova seção sobre implementações que antecede os exercícios e as notas bibliográficas, no final de cada capítulo. Além da expansão e da revisão do texto original, foram incluídos os capítulos Caminhos Mínimos e Emparelhamentos Máximos em Grafos, e a seção Complexidade de um Problema Numérico no capítulo Problemas NP-Completos.

O enfoque original, que situa a Teoria dos Grafos dentro da área de Combinatória Computacional, é aprofundado de modo rigoroso e coerente, através da revisão e da inclusão de capítulos e seções onde o estudo dos grafos é realizado através dos seus algoritmos, cujo desenvolvimento, análise de corretude e de complexidade, e implementação são considerados.

Aqui lembramos também de um coautor do Jayme, o famoso pesquisador Donald E. Knuth, que fundamentou a análise rigorosa da complexidade computacional de algoritmos: "Ciência é o que nós entendemos bem o suficiente para explicar para um computador, e Arte é todo o resto que nós fazemos." Nosso professor emérito Jayme Luiz Szwarcfiter nos ensina como a Matemática transforma arte em ciência, e o seu livro é inspiração e incentivo a todos aqueles que acreditam na universidade brasileira.

**Celina M. Herrera de Figueiredo**  
PESC/COPPE  
Universidade Federal do Rio de Janeiro (UFRJ)

Página deixada intencionalmente em branco

---

# Apresentação às Implementações

---

Na reformulação deste livro, além da ampliação do conjunto de algoritmos estudados, implementações dos algoritmos foram incluídas ao final de cada capítulo. Apesar do estudo dos algoritmos através de linguagens em pseudocódigo possuir as vantagens de evidenciar o método de solução e omitir detalhes operacionais requeridos pelas linguagens de programação, a apresentação de implementações é uma importante complementação. Os benefícios de se dispor de implementações se estendem tanto do ponto de vista didático e acadêmico, quanto profissional. No primeiro grupo, estudantes, docentes e pesquisadores, com objetivos diferentes, podem usufruir dos resultados obtidos pelos programas. No segundo grupo, programadores, analistas e profissionais da computação, de um modo geral, podem aplicar diretamente a implementação já pronta e descrita neste texto.

As implementações são apresentadas como última seção de cada capítulo de modo a permitir que sejam consultadas seletivamente. Tal separação é conveniente tanto ao leitor interessado nas discussões pertinentes à implementação, como àqueles com interesse puramente teórico nos algoritmos.

A linguagem escolhida para as implementações foi Python, em sua versão 3.4.3, recente no momento da preparação deste livro. Python é uma linguagem cuja utilização está em larga expansão, tanto na indústria quanto na academia, sendo utilizada modernamente como linguagem introdutória à computação nas mais importantes universidades. Pode-se explicar o sucesso de Python por sua sintaxe simples e, ao mesmo tempo, de grande poder de expressão. Esta última característica é em particular conveniente para o propósito de diminuir a distância entre o pseudocódigo e o programa. Dada sua popularidade, supõe-se neste texto que Python seja uma linguagem familiar ao leitor. Se este não for o caso, o subconjunto de recursos utilizados de Python é intencionalmente mantido pequeno e, com a ajuda de algum texto de referência que introduza a linguagem, pode-se facilmente acompanhar as implementações. Tais implementações são apresentadas de forma similar aos respectivos algoritmos em pseudocódigo, procurando manter tanto quanto possível um mapeamento "um-para-um" entre as abstrações (comandos, variáveis e funções, dentre outros elementos lógicos).

É importante observar que os algoritmos e suas implementações são relativamente independentes. No entanto, alguns algoritmos servem de base para outros, como é o caso das buscas em grafos e dos algoritmos de manipulação da representação de grafos. Tais algoritmos são apresentados apenas na sua primeira ocorrência, por simplicidade.

Todas as implementações em Python passaram por extensivos testes, cujos resultados foram comparados com os de implementações independentes feitas em C/C++, o que lhes confere razoável confiabilidade.

Em resumo, acreditamos que o acréscimo das implementações nesta reformulação do livro agrega o importante aspecto prático à excelente exposição teórica de Jayme. Num nível pessoal, não podemos deixar de agradecê-lo pelo trabalho em conjunto. Dadas suas qualidades bem conhecidas, nas quais se destacam seu conhecimento, criatividade, disciplina e bom humor, o convívio com ele neste tempo foi enriquecedor e tornaram a jornada excitante.

**Fabiano de Souza Oliveira  
Paulo Eustáquio Duarte Pinto**  
Instituto de Matemática e Estatística  
Universidade do Estado do Rio de Janeiro (UERJ)

Página deixada intencionalmente em branco

---

# Sumário

---

<b>Notação</b>	<b>xvii</b>
<b>Índice de Algoritmos</b>	<b>xix</b>
<b>Índice de Programas</b>	<b>xix</b>
<b>Índice de Figuras</b>	<b>xxi</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Breve Descrição do Conteúdo . . . . .	1
1.2 Os Grafos: Um Pouco de História . . . . .	2
1.3 Apresentação dos Algoritmos . . . . .	4
1.4 Complexidade de Algoritmos . . . . .	6
1.5 Estruturas de Dados . . . . .	10
1.6 A Linguagem Python . . . . .	12
1.7 Implementações . . . . .	15
1.7.1 Algoritmo 1.3: Ordenação de Sequências . . . . .	15
1.8 Exercícios . . . . .	15
1.9 Notas Bibliográficas . . . . .	17
<b>2 Uma Iniciação à Teoria dos Grafos</b>	<b>19</b>
2.1 Introdução . . . . .	19
2.2 Os Primeiros Conceitos . . . . .	19
2.3 Árvores . . . . .	24
2.4 Conectividade . . . . .	30
2.5 Planaridade . . . . .	32
2.6 Ciclos Hamiltonianos . . . . .	35
2.7 Coloração . . . . .	36
2.8 Grafos Direcionados . . . . .	38
2.9 Representação de Grafos . . . . .	40

2.10	Implementações em Python: Operações Básicas . . . . .	42
2.10.1	Algoritmo 2.1: Centro de Árvore . . . . .	47
2.11	Exercícios . . . . .	47
2.12	Notas Bibliográficas . . . . .	51
<b>3</b>	<b>Técnicas Básicas</b>	<b>53</b>
3.1	Introdução . . . . .	53
3.2	Processo de Representação . . . . .	53
3.3	Adjacências . . . . .	54
3.4	Ordenação de Vértices ou Arestas . . . . .	54
3.5	Coloração Aproximada . . . . .	55
3.6	Ordenação Topológica . . . . .	57
3.7	Recursão . . . . .	58
3.8	Árvores de Decisão . . . . .	58
3.9	Limite Inferior para Ordenação . . . . .	59
3.10	Programas em Python . . . . .	59
3.10.1	Algoritmo 3.3: Coloração Aproximada . . . . .	59
3.10.2	Algoritmo 3.4: Ordenação Topológica . . . . .	60
3.11	Exercícios . . . . .	60
3.12	Notas Bibliográficas . . . . .	62
<b>4</b>	<b>Buscas em Grafos</b>	<b>63</b>
4.1	Introdução . . . . .	63
4.2	Algoritmo Básico . . . . .	64
4.3	Busca em Profundidade . . . . .	66
4.4	Biconectividade . . . . .	68
4.5	Busca em Profundidade – Digrafos . . . . .	70
4.6	Componentes Fortemente Conexas . . . . .	74
4.7	Busca em Largura . . . . .	76
4.8	Busca em Largura Lexicográfica . . . . .	79
4.9	Reconhecimento dos Grafos Cordais . . . . .	82
4.10	Busca Irrestrita . . . . .	86
4.11	Programas em Python . . . . .	89
4.11.1	Algoritmo 4.2: Busca em Profundidade . . . . .	89
4.11.2	Algoritmo 4.4: Busca em Profundidade em Digrafos . . . . .	90
4.11.3	Algoritmo 4.5: Componentes Fortemente Conexas . . . . .	90
4.11.4	Algoritmo 4.6: Busca em Largura . . . . .	91
4.11.5	Algoritmo 4.7: Busca em Largura Lexicográfica . . . . .	91
4.11.6	Algoritmo 4.8: Busca Irrestrita . . . . .	92
4.12	Exercícios . . . . .	92
4.13	Notas Bibliográficas . . . . .	99

<b>5</b>	<b>Outras Técnicas</b>	<b>101</b>
5.1	Introdução . . . . .	101
5.2	Algoritmo Guloso . . . . .	101
5.3	Árvore Geradora Mínima . . . . .	102
5.4	Programação Dinâmica . . . . .	105
5.5	Particionamento de Árvores . . . . .	106
5.6	Alteração Estrutural . . . . .	112
5.7	Número Cromático . . . . .	113
5.8	Programas em Python . . . . .	116
5.8.1	Algoritmo 5.1: Árvore Geradora Mínima . . . . .	116
5.8.2	Algoritmo 5.2: Particionamento de Árvore . . . . .	117
5.8.3	Algoritmo 5.3: Número Cromático . . . . .	118
5.9	Exercícios . . . . .	118
5.10	Notas Bibliográficas . . . . .	120
<b>6</b>	<b>Fluxo Máximo em Redes</b>	<b>123</b>
6.1	Introdução . . . . .	123
6.2	O Problema do Fluxo Máximo . . . . .	123
6.3	O Teorema do Fluxo Máximo – Corte Mínimo . . . . .	126
6.4	Um Primeiro Algoritmo . . . . .	128
6.5	Um Algoritmo $O(nm^2)$ . . . . .	129
6.6	Um Algoritmo $O(n^2m)$ . . . . .	130
6.7	Um Algoritmo $O(n^3)$ . . . . .	133
6.8	Programas em Python . . . . .	136
6.8.1	Algoritmo 6.1: Fluxo Máximo . . . . .	136
6.8.2	Algoritmo 6.3: Fluxo Máximo - Rede de Camadas . . . . .	137
6.9	Exercícios . . . . .	138
6.10	Notas Bibliográficas . . . . .	141
<b>7</b>	<b>Caminhos Mínimos</b>	<b>143</b>
7.1	Introdução . . . . .	143
7.2	As Equações de Bellman . . . . .	144
7.3	Algoritmo de Dijkstra . . . . .	144
7.4	O Algoritmo de Bellman-Ford . . . . .	148
7.5	O Algoritmo de Floyd . . . . .	149
7.6	$k$ -ésimos Caminhos Mínimos . . . . .	151
7.7	$k$ -ésimos Caminhos Mínimos Simples . . . . .	156
7.8	Detecção de ciclos negativos . . . . .	159
7.9	Programas em Python . . . . .	162
7.9.1	Algoritmo 7.1: Dijkstra . . . . .	162
7.9.2	Algoritmo 7.2: Bellman–Ford . . . . .	162

7.9.3	Algoritmo 7.3: Floyd . . . . .	163
7.9.4	Algoritmo 7.4: $k$ -ésimo mínimo (passeio) . . . . .	163
7.9.5	Algoritmo 7.5: $k$ -ésimo mínimo (caminho simples) . . . . .	164
7.9.6	Algoritmo 7.6: Detecção de Ciclos Negativos . . . . .	165
7.10	Exercícios . . . . .	166
7.11	Notas Bibliográficas . . . . .	170
<b>8</b>	<b>Emparelhamentos Máximos em Grafos</b>	<b>171</b>
8.1	Introdução . . . . .	171
8.2	Emparelhamentos Perfeitos . . . . .	172
8.3	Caminhos Alternantes . . . . .	173
8.4	Grafos Bipartidos sem Pesos: Cardinalidade Máxima . . . . .	174
8.5	O Método Húngaro . . . . .	176
8.6	Emparelhamentos e Coberturas por Vértices . . . . .	180
8.7	Grafos Bipartidos Ponderados . . . . .	181
8.8	Grafos Gerais sem Peso . . . . .	184
8.9	Programas em Python . . . . .	191
8.9.1	Algoritmo 8.1: Emparelhamento Bipartido (com Digrafo) . . . . .	191
8.9.2	Algoritmo 8.2: Emparelhamento Bipartido (Método Húngaro) . . . . .	193
8.9.3	Algoritmo 8.3: Emparelhamento Bipartido Ponderado . . . . .	194
8.9.4	Algoritmo 8.4: Emparelhamento Geral . . . . .	195
8.10	Exercícios . . . . .	198
8.11	Notas Bibliográficas . . . . .	201
<b>9</b>	<b>Problemas NP-Completos</b>	<b>203</b>
9.1	Introdução . . . . .	203
9.2	Problemas de Decisão . . . . .	203
9.3	A Classe P . . . . .	206
9.4	Alguns Problemas Aparentemente Difíceis . . . . .	207
9.5	A Classe NP . . . . .	210
9.6	A Questão $P = NP$ . . . . .	213
9.7	Complementos de Problemas . . . . .	214
9.8	Transformações Polinomiais . . . . .	215
9.9	Alguns Problemas NP-Completos . . . . .	217
9.10	Restrições e Extensões de Problemas . . . . .	220
9.11	Algoritmos Pseudopolinomiais . . . . .	224
9.12	Complexidade de um Problema Numérico . . . . .	227
9.13	Programas em Python . . . . .	231
9.13.1	Algoritmo 9.1: Subconjunto Soma . . . . .	231
9.14	Exercícios . . . . .	231
9.15	Notas Bibliográficas . . . . .	234

<b>Referências</b>	<b>235</b>
<b>Posfácio</b>	<b>244</b>
<b>Índice</b>	<b>246</b>

Página deixada intencionalmente em branco

---

# Notação

---

## Conjuntos

$s \in S$	$s$ é elemento do conjunto $S$ .
$S \subseteq R$	conjunto $S$ é subconjunto do conjunto $R$ .
$S \subset R$	conjunto $S$ é subconjunto próprio do conjunto $R$ .
$\emptyset$	conjunto vazio.
$S \cup R$	união dos conjuntos $S$ e $R$ .
$S \cap R$	interseção dos conjuntos $S$ e $R$ .
$S - R$	diferença do conjunto $S$ para $R$ .
$S \times R$	produto cartesiano dos conjuntos $S$ e $R$ .
$ S $	cardinalidade do conjunto $S$ .
$S \Delta R$	diferença simétrica entre os conjuntos $S$ e $R$ .

## Funções

$f : S \rightarrow R$	$f$ é uma função de domínio $S$ e contradomínio $R$ .
$f(s)$	valor da função $f$ para o argumento $s$ .
$f(S)$	imagem do conjunto $S$ pela função $f$ .
$O(f)$	função assintoticamente dominada por $f$ .
$\Omega(f)$	função que assintoticamente domina $f$ .

## Números

$ n $	valor absoluto do número $n$ .
$\lfloor n \rfloor$	maior inteiro $\leq n$ .
$\lceil n \rceil$	menor inteiro $\geq n$ .
$n!$	fatorial de $n$ .
$\log n$	logaritmo* (base 2) de $n$ .
$\binom{n}{k}$	número de combinações de $n$ elementos $k$ a $k$ .

## Grafos

$G(V, E)$	grafo $G$ com conjunto de vértices $V$ e arestas $E$ .
$(v, w) \in E$	aresta $(v, w)$ do conjunto $E$ .
$G - s$	grafo obtido de $G$ pela remoção do vértice (aresta) $s$ .
$G - S$	grafo obtido de $G$ pela remoção do conjunto de vértices (arestas) $S$ .
$G + s$	grafo obtido de $G$ pela inclusão do vértice (aresta) $s$ .

\*Todos os logaritmos considerados no texto são da base 2.

$G + S$	grafo obtido de $G$ pela inclusão do conjunto de vértices (arestas) $S$ .
$\overline{G}$	complemento do grafo $G$ .
$K_n$	grafo completo com $n$ vértices.
$P_n$	caminho induzido com $n$ vértices.
$C_n$	ciclo induzido com $n$ vértices.
$K_{n,m}$	grafo bipartido $(V_1 \cup V_2, E)$ completo, $ V_1  = n$ e $ V_2  = m$ .
$T_v$	subárvore de raiz $v$ da árvore enraizada $T$ .
$A(v)$	lista de adjacências do vértice $v$ .
$\chi(G)$	número cromático do grafo $G$ .

---

# Índice de Algoritmos

---

1.1	Inversão de uma sequência . . . . .	6
1.2	Ordenação de uma sequência (básico) . . . . .	9
1.3	Ordenação de uma sequência . . . . .	10
2.1	Determinação do centro de uma árvore . . . . .	27
3.1	Estrutura de adjacências de um digrafo . . . . .	53
3.2	Coloração aproximada . . . . .	55
3.3	Coloração aproximada . . . . .	56
3.4	Ordenação topológica . . . . .	57
4.1	Busca geral . . . . .	65
4.2	Busca em profundidade (utilizando pilha) . . . . .	66
4.3	Busca em profundidade (recursivo) . . . . .	67
4.4	Busca em profundidade em digrafos . . . . .	71
4.5	Componentes fortemente conexas de um digrafo . . . . .	76
4.6	Busca em largura . . . . .	77
4.7	Busca em largura lexicográfica . . . . .	81
4.8	Busca irrestrita em profundidade (utilizando pilha) . . . . .	87
4.9	Busca irrestrita em profundidade (recursivo) . . . . .	89
5.1	Árvore geradora mínima . . . . .	104
5.2	Particionamento de árvores . . . . .	111
5.3	Determinação do número cromático $\chi$ de um grafo . . . . .	114
6.1	Fluxo máximo em uma rede (Ford e Fulkerson) . . . . .	128
6.2	Construção da rede de camadas $D^*(f)$ . . . . .	130
6.3	Fluxo máximo em uma rede (Dinic) . . . . .	132
7.1	Caminhos mínimos – Dijkstra . . . . .	145
7.2	Caminhos mínimos: Bellman-Ford . . . . .	149
7.3	Caminhos mínimos: Floyd . . . . .	150
7.4	$k$ -ésimos caminhos mínimos . . . . .	155
7.5	$k$ -ésimos caminhos simples mínimos . . . . .	158
7.6	Detecção de ciclos negativos . . . . .	161
8.1	Emparelhamento cardinalidade máxima – Grafos bipartidos . . . . .	175
8.2	Emparelhamento cardinalidade máxima – Método Húngaro . . . . .	179
8.3	Emparelhamento ponderado máximo – Grafos bipartidos . . . . .	184
8.4	Emparelhamento em grafos gerais . . . . .	191
9.1	Subconjunto Soma . . . . .	230

Página deixada intencionalmente em branco

---

# Índice de Programas

---

1.1	Inversão de uma sequência . . . . .	14
1.2	Ordenação de sequências . . . . .	15
2.1	Determinação do centro de uma árvore . . . . .	47
3.1	Coloração aproximada . . . . .	59
3.2	Ordenação topológica . . . . .	60
4.1	Busca em profundidade (utilizando pilha) . . . . .	89
4.2	Busca em profundidade em digrafos . . . . .	90
4.3	Componentes fortemente conexas de um digrafo . . . . .	90
4.4	Busca em largura . . . . .	91
4.5	Busca em largura lexicográfica . . . . .	91
4.6	Busca irrestrita em profundidade (utilizando pilha) . . . . .	92
5.1	Árvore geradora mínima . . . . .	116
5.2	Particionamento de árvores . . . . .	117
5.3	Determinação do número cromático $\chi$ de um grafo . . . . .	118
6.1	Fluxo máximo em uma rede (Ford e Fulkerson) . . . . .	136
6.2	Fluxo máximo em uma rede de camadas . . . . .	137
7.1	Caminhos mínimos - Dijkstra . . . . .	162
7.2	Caminhos mínimos - Bellman-Ford . . . . .	162
7.3	Caminhos mínimos - Floyd . . . . .	163
7.4	k-ésimos caminhos mínimos . . . . .	163
7.5	k-ésimos caminhos simples mínimos . . . . .	164
7.6	Detecção de ciclos negativos . . . . .	165
8.1	Emparelhamento cardinalidade máxima - Grafos bipartidos . . . . .	192
8.2	Emparelhamento cardinalidade máxima - Método Húngaro . . . . .	193
8.3	Emparelhamento ponderado máximo - Grafos bipartidos . . . . .	194
8.4	Emparelhamento em grafos gerais . . . . .	195
9.1	Subconjunto Soma . . . . .	231

Página deixada intencionalmente em branco

---

# Lista de Figuras

---

1.1	O problema da ponte de Königsberg . . . . .	2
1.2	Quatro cores são necessárias . . . . .	3
1.3	Um algoritmo . . . . .	4
1.4	Linhas de um algoritmo . . . . .	5
1.5	Um modelo computacional . . . . .	7
1.6	Representações diferentes de uma lista . . . . .	11
1.7	Uma lista duplamente encadeada . . . . .	11
1.8	Pilha e fila . . . . .	12
2.1	Um grafo $(V, E)$ e uma representação geométrica do mesmo . . . . .	20
2.2	Grafos isomorfos e não isomorfos . . . . .	20
2.3	Um grafo com laços e um multigrafo . . . . .	20
2.4	Um grafo desconexo . . . . .	21
2.5	Operações de exclusão e inclusão de arestas ou vértices . . . . .	22
2.6	O passo principal do Teorema 2.1 . . . . .	22
2.7	Um grafo e seu complemento . . . . .	23
2.8	Grafos completos . . . . .	23
2.9	Exemplos de grafo bipartido . . . . .	23
2.10	Exemplos de subgrafos . . . . .	24
2.11	Uma floresta composta de duas árvores . . . . .	25
2.12	As árvores com 6 vértices . . . . .	25
2.13	Excentricidade de vértices . . . . .	26
2.14	Determinação do centro de uma árvore . . . . .	27
2.15	Subgrafo gerador e árvore geradora . . . . .	27
2.16	Árvores . . . . .	28
2.17	Subárvore e subárvore parcial . . . . .	28
2.18	Árvores enraizadas isomórficas . . . . .	29
2.19	Árvores $m$ -árias . . . . .	29
2.20	Cortes de um grafo . . . . .	30
2.21	Um grafo e seus blocos . . . . .	31
2.22	Prova do Lema 2.3 . . . . .	31
2.23	Um grafo planar . . . . .	33
2.24	Imersão de um grafo planar na esfera . . . . .	33
2.25	Os grafos $K_5$ e $K_{3,3}$ . . . . .	34

2.26	Subdivisão de arestas . . . . .	34
2.27	Um grafo biconexo não hamiltoniano . . . . .	35
2.28	Prova do Teorema 2.10 . . . . .	36
2.29	Coloração de grafos . . . . .	36
2.30	Grafos $k$ -críticos . . . . .	37
2.31	Coloração de mapas . . . . .	37
2.32	Um digrafo e seu grafo subjacente . . . . .	38
2.33	Digrafos unilateralmente e fracamente conexos . . . . .	38
2.34	Um digrafo, seu fecho e redução transitivos . . . . .	39
2.35	Uma árvore direcionada enraizada . . . . .	40
2.36	Matrizes de adjacências . . . . .	40
2.37	Matriz de incidências . . . . .	41
2.38	Estrutura de adjacência . . . . .	41
2.39	Lista de adjacência . . . . .	45
2.40	O grafo de Petersen . . . . .	48
3.1	Ordenação por caixas . . . . .	54
3.2	Colorações aproximada (a) e ótima (b) . . . . .	56
3.3	Exemplo para o Algoritmo 3.3 . . . . .	57
3.4	Árvore de decisão . . . . .	58
4.1	Uma árvore enraizada . . . . .	64
4.2	Como caminhar sistematicamente? . . . . .	64
4.3	Exemplo para uma busca . . . . .	65
4.4	Busca em profundidade . . . . .	68
4.5	Outra busca em profundidade . . . . .	69
4.6	Determinação das componentes biconexas de um grafo . . . . .	70
4.7	Busca em profundidade em digrafos . . . . .	73
4.8	Busca em profundidade completa em digrafos. Caso geral . . . . .	74
4.9	Componentes fortemente conexas do digrafo da Figura 4.8(a) . . . . .	77
4.10	Tipos de arestas na busca em largura . . . . .	78
4.11	Uma busca em largura . . . . .	80
4.12	Busca em largura lexicográfica e não lexicográfica . . . . .	81
4.13	Busca em largura lexicográfica com rotulação de vértices . . . . .	82
4.14	Grafo não cordal . . . . .	83
4.15	Prova do Teorema 4.7 . . . . .	83
4.16	Prova do Teorema 4.9 . . . . .	84
4.17	Prova do Lema 4.10 . . . . .	85
4.18	Reconhecimento de esquemas de eliminação perfeita . . . . .	85
4.19	Correção: esquemas de eliminação perfeita . . . . .	86
4.20	Ciclicidade . . . . .	88
4.21	Busca irrestrita em profundidade de raiz $a$ . . . . .	88
4.22	O subgrafo proibido . . . . .	95
5.1	Um grafo com pesos nas arestas . . . . .	103
5.2	Algoritmo de árvore geradora mínima . . . . .	103

5.3	Prova do Lema 5.2 . . . . .	104
5.4	Sequência de Fibonacci, para $n \leq 9$ . . . . .	105
5.5	Um particionamento de uma árvore . . . . .	106
5.6	Um particionamento ótimo . . . . .	107
5.7	Construção de $P(v, j)$ . . . . .	108
5.8	Construção de $P(v, 3)$ , sendo $k = 4$ . . . . .	109
5.9	Dados para o cálculo de $p(v, 3)$ . . . . .	109
5.10	Cálculo de $p(v, 3)$ . . . . .	109
5.11	A subárvore parcial $T_v^i$ . . . . .	110
5.12	O exemplo correspondente à Figura 5.13 . . . . .	112
5.13	Tabela de solução do problema da Figura 5.12, com $k = 3$ . . . . .	112
5.14	Operação de condensação . . . . .	113
5.15	Os grafos $\alpha$ e $\beta$ . . . . .	114
5.16	Determinação do número cromático . . . . .	115
6.1	Fluxo em redes . . . . .	124
6.2	Fluxos maximal e máximo . . . . .	125
6.3	Prova do Lema 6.1 . . . . .	126
6.4	Redes residuais . . . . .	127
6.5	Um caso ruim para o Algoritmo 6.1 . . . . .	129
6.6	Esquema de uma rede de camadas . . . . .	130
6.7	Rede de camadas para o fluxo da Figura 6.1(a) . . . . .	131
6.8	Entrada para o exemplo da Figura 6.9 . . . . .	132
6.9	Um exemplo do Algoritmo 6.3 . . . . .	133
6.10	Um exemplo do algoritmo de fluxo maximal . . . . .	135
6.11	Sugestão do Exercício 6.5 . . . . .	139
7.1	Um grafo e sua matriz de distâncias . . . . .	144
7.2	Árvore de caminhos mínimos . . . . .	145
7.3	Exemplo para o algoritmo de Dijkstra . . . . .	147
7.4	Teorema 7.1 . . . . .	147
7.5	Cálculo de $c(\ell, k)$ . . . . .	149
7.6	Exemplo para o algoritmo de Bellman-Ford . . . . .	149
7.7	Matriz $W_{k-1}$ . . . . .	151
7.8	Exemplo para o algoritmo de Floyd . . . . .	151
7.9	Computação das matrizes da Figura 7.8 . . . . .	152
7.10	Exemplo para desvios . . . . .	152
7.11	Equação (*) . . . . .	154
7.12	Exemplo para o Algoritmo 7.4 . . . . .	155
7.13	Desvio simples relativo a $C_i(j)$ . . . . .	157
7.14	O grafo $D^+$ . . . . .	160
7.15	Exemplo para Algoritmo 7.6 . . . . .	161
8.1	Emparelhamentos maximal e máximo . . . . .	171
8.2	Emparelhamento perfeito . . . . .	172
8.3	Caminhos alternante e aumentante . . . . .	173

8.4	Aplicação do Lema 8.1 . . . . .	175
8.5	Exemplo de aplicação do Algoritmo 8.1 . . . . .	176
8.6	Floresta $M$ -alternante . . . . .	177
8.7	Floresta húngara . . . . .	178
8.8	Emparelhamentos e cobertura ponderados . . . . .	181
8.9	Um exemplo de aplicação do Algoritmo 8.3 . . . . .	185
8.10	Exemplo de emparelhamentos em grafo não bipartido . . . . .	186
8.11	Ciclos par (a) e ímpar (b) em passeios . . . . .	187
8.12	Grafo direcionado $D_f(M)$ da Figura 8.10(a) . . . . .	190
8.13	Grafo $G$ . . . . .	198
9.1	Instância do problema de clique . . . . .	204
9.2	Problemas de decisão, localização e otimização associados . . . . .	205
9.3	Problema do caixeiro viajante . . . . .	206
9.4	Conjunção e disjunção . . . . .	208
9.5	Exemplo para os problemas 2, 3 e 4 . . . . .	208
9.6	Exemplos para os problemas 5, 7, 8 . . . . .	209
9.7	Exemplo para o problema 10 . . . . .	210
9.8	Justificativa para resposta ao problema <i>Ciclo Hamiltoniano</i> . . . . .	211
9.9	Conjecturas $P \neq NP$ , $NP \neq Co-NP$ , $P \neq NP \cap Co-NP$ . . . . .	216
9.10	Transformação polinomial do problema $\Pi_1$ em $\Pi_2$ . . . . .	216
9.11	Equivalência entre $\Pi_1(D_1, Q_1)$ e $(f(D_1), Q_2)$ . . . . .	217
9.12	Transformação da prova do Teorema 9.2 . . . . .	219
9.13	Vértices de $G$ . . . . .	222
9.14	Arestas adicionais de $G$ . . . . .	223
9.15	Grafo da prova do Teorema 9.7 . . . . .	223
9.16	Contradição na prova do Lema 9.4. . . . .	224
9.17	Atribuição de cores do Lema 9.7. . . . .	225
9.18	Grafo $G$ e sua matriz de incidências $B$ . . . . .	228
9.19	Matriz $B'$ , dos elementos de $S$ . . . . .	228
9.20	Matriz $X$ do exemplo . . . . .	230

Página deixada intencionalmente em branco

# INTRODUÇÃO

---

## 1.1 Breve Descrição do Conteúdo

Este texto corresponde a um curso introdutório de algoritmos e grafos. A importância do assunto no campo teórico se reflete na grande quantidade de problemas existentes na área, ainda em processo de estudo. Além disso, no acentuado crescimento do número de publicações e artigos especializados e na quantidade de grupos de pesquisa em universidades que se dedicam a esse estudo. Trata-se de matéria atual, ainda em formação. Por outro lado, é um tema relevante do ponto de vista prático, tanto em computação quanto em matemática aplicada. São inúmeros os problemas práticos que podem ser resolvidos mediante uma modelagem em grafos e posterior utilização de um programa de computador, implementando um algoritmo conveniente.

O assunto é abordado sob um enfoque computacional, ou seja, os algoritmos são desenvolvidos considerando-se uma posterior implementação em computador. Nesse sentido, a eficiência de tempo e espaço de um processo são fatores básicos para sua avaliação. Simultaneamente, o tema é apresentado sob um tratamento matemático. Este tratamento pode ser percebido, por exemplo, na verificação de correção de um algoritmo, ou na determinação de sua eficiência. Contudo, não se requer do leitor conhecimentos especializados nessa área para acompanhar o texto.

O Capítulo 1 contém conceitos básicos de algoritmos, bem como uma descrição da notação a ser utilizada no texto. Além disso, uma breve descrição da linguagem Python, empregada na implementação dos algoritmos. O Capítulo 2, por sua vez, contém uma introdução amigável à Teoria de Grafos, bem como a notação de seus conceitos utilizados no decorrer do texto. A apresentação dos algoritmos se desenvolve principalmente em torno da descrição de técnicas gerais, utilizadas na sua formulação. Após a exposição de cada técnica segue-se uma aplicação correspondente, ou seja, um algoritmo desenvolvido mediante o uso da técnica em questão. Para cada um desses algoritmos, é efetuado um estudo de sua validade e determinada a sua complexidade.

No espírito acima são apresentados os Capítulos 3, 4 e 5, onde técnicas em grafos e algoritmos de aplicação encontram-se entremeados. O Capítulo 3 descreve várias técnicas elementares. O seguinte contém um estudo detalhado de busca em grafos—métodos dos mais empregados em problemas algorítmicos. O Capítulo 5 inclui a apresentação de técnicas utilizadas em otimização combinatória. O Capítulo 6 é dedicado ao problema do fluxo máximo. Sua solução emprega diversos conceitos examinados no decorrer do texto. Além disso, é uma ilustração didática do estudo de complexidade. O Capítulo 7 é dedicado ao importante problema da determinação de caminhos mínimos em grafos, onde diversas variantes do problema são apresentadas. O Capítulo 8 contém algoritmos para resolver o problema do emparelhamento em grafos. O último capítulo descreve os princípios da teoria do NP-completo. Sua apresentação se desenvolve de modo a prescindir do leitor conhecimentos especializados em teoria da computação. A formulação utilizada representa uma tentativa de tornar mais simples a compreensão do assunto, preservando, contudo, o seu caráter matemático. A vasta aplicação da teoria do NP-completo em algoritmos para grafos torna quase obrigatória a sua inclusão em um livro desta natureza.

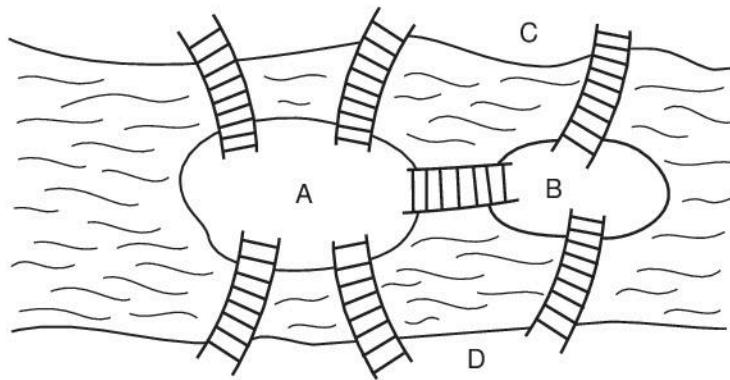


Figura 1.1: O problema da ponte de Königsberg

Após cada capítulo, é apresentada uma lista de exercícios, de dificuldade variada. O objetivo, naturalmente, é auxiliar o estudante no aprendizado. Em alguns capítulos, os últimos exercícios foram retirados de concursos, exames de capacitação ou universidades. Em particular, aqueles referenciados como UVA são originários da Universidade de Valladolid, Espanha.

A bibliografia referenciada, em geral, é mais abrangente do que o texto. Contudo, este fato é proveitoso para leitores que desejam se aprofundar nos temas abordados. Além disso, pode ser utilizada como fonte para pesquisa bibliográfica na área. Ênfase especial foi dada à bibliografia em língua portuguesa.

Conforme mencionado, além da descrição dos algoritmos, o livro contém suas implementações na linguagem Python. Assim, o texto pode ser utilizado tanto pelo leitor interessado nos aspectos mais teóricos desses processos, como por aquele que visa a utilização prática dos algoritmos. Todas as implementações foram simuladas e testadas com entradas diversas.

## 1.2 Os Grafos: Um Pouco de História

De um modo geral, a área de algoritmos em grafos pode ser caracterizada como aquela cujo interesse principal é resolver problemas algorítmicos em grafos, tendo em mente uma preocupação computacional. Ou seja, o objetivo principal é encontrar algoritmos, eficientes se possível, para resolver um dado problema em grafos. Naturalmente, existem diferenças entre essa área e a teoria de grafos. Em primeiro lugar, há problemas não algorítmicos em grafos em que o conceito de eficiência torna-se sem sentido. Por outro lado, a preocupação pela eficiência, na área de algoritmos, se traduz também na formulação de problemas algorítmicos em grafos que talvez se encontrem fora do interesse para a teoria. Não obstante, é bastante comum o fato de que uma possível melhora na eficiência de um algoritmo para um dado problema em grafos somente pode ser obtida através de um maior conhecimento teórico do problema, ou seja, algoritmos mais eficientes, em geral, significam a utilização de processos, cuja correção é dependente de aspectos de interesse teórico em grafos.

O assunto que se constituiu no marco inicial da teoria de grafos é na realidade um problema algorítmico. Além disso, é um problema cuja solução foi a elaboração de um algoritmo eficiente. É o conhecido *problema da ponte de Königsberg* resolvido por Euler em 1736. No Rio Pregel, junto à cidade de Königsberg (hoje Kaliningrado) na então Prússia, existem duas ilhas formando portanto quatro regiões distinguíveis da terra, A, B, C e D. Há um total de sete pontes interligando-as, conforme a disposição indicada na Figura 1.1. O problema consiste em, partindo de uma dessas regiões, determinar um trajeto pelas pontes segundo o qual se possa retornar à região de partida, após atravessar cada ponte exatamente uma vez. Euler mostrou que não existe tal trajeto, ao utilizar um modelo em grafos para uma generalização deste problema. Através desse modelo ele verificou que existe o desejado trajeto quando e somente quando em cada região concorrer um número par de pontes.

A solução desse problema é considerada o primeiro teorema em teoria de grafos. Embora tenha sido estabelecida há mais de 200 anos, muito pouco foi realizado nos anos próximos subsequentes ao trabalho de Euler. Por volta de meados do século XIX, três desenvolvimentos isolados contribuiriam enormemente para despertar o

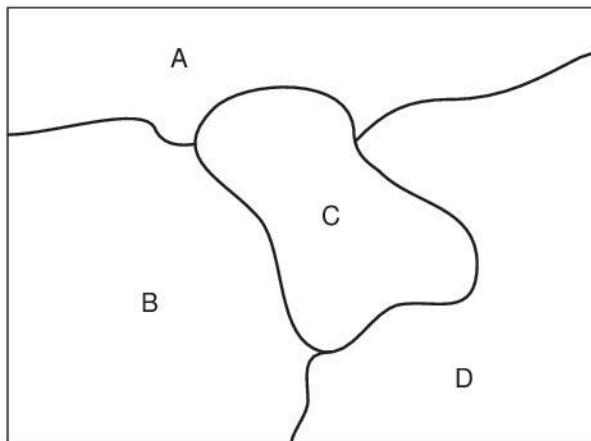


Figura 1.2: Quatro cores são necessárias

interesse pela área. O primeiro é a formulação do *problema das quatro cores*. Supõe-se que a autoria do problema seja de Francis Guthrie. Este, em 1852, o teria proposto a seu irmão Frederick, então estudante, que por sua vez o comunicou a De Morgan. Outro desenvolvimento importante foi a formulação do *problema do ciclo Hamiltoniano*, por Hamilton. Este problema deu origem, na ocasião, a um quebra-cabeça, o qual foi inclusive comercializado. O terceiro acontecimento marcante do século XIX foi o desenvolvimento da *teoria das árvores*, realizado, inicialmente, por Kirchhoff e por Cayley. O primeiro visava a sua aplicação em circuitos elétricos, enquanto que o último a empregava em química orgânica. As árvores constituem uma classe especial de grafos, com larga aplicação nas mais diferentes áreas.

O problema das quatro cores consiste em colorir os países de um mapa arbitrário plano, cada país com uma cor, de tal forma que países fronteiriços possuam cores diferentes. O problema então consiste em obter tal coloração usando não mais de 4 cores. É simples apresentar um exemplo de um mapa, como o da Figura 1.2, onde 3 cores não são suficientes. Por outro lado, foi formulada uma prova de que 5 o são. Conjeturou-se então que 4 cores também seriam suficientes. Esta conjectura permaneceu em aberto até 1977, quando foi provada por Appel e Haken. Além da importância do tópico de coloração, o problema das 4 cores desempenhou um papel muito relevante para o desenvolvimento geral da teoria dos grafos, pois serviu de motivação para o trabalho na área e ensejou o desenvolvimento de outros aspectos teóricos, realizados na tentativa de resolver a questão.

No problema do caminho hamiltoniano existem  $n$  cidades. Cada par de cidades pode ser adjacentes ou não, arbitrariamente. Partindo de uma cidade qualquer, o problema consiste em determinar um trajeto que passe exatamente uma vez em cada cidade e retorne ao ponto de partida, e tal que cada par de cidades consecutivas no trajeto seja sempre adjacente. Uma solução de força bruta consiste, por exemplo, em examinar cada permutação do conjunto das cidades e verificar se esta corresponde ou não a um trajeto com as condições exigidas. Essa solução não é satisfatória do ponto de vista algorítmico, pois apenas no caso em que  $n$  é muito pequeno torna-se possível examinar as  $n!$  permutações. Até a data atual, não foi encontrada uma solução algorítmica satisfatória, ou seja, não são conhecidas condições necessárias e suficientes, razoáveis, de existência de tais trajetos.

No século XX o interesse pelos grafos aumentou. Por volta da década de 1930, resultados fundamentais na teoria foram obtidos por Kuratowski, König e Menger entre outros. Os anos mais recentes confirmam, de certa forma, a ideia de ser a teoria de grafos uma área ainda com vastas regiões inexploradas.

O outro elemento principal desse texto é o *algoritmo*. Ele pode ser associado ao desenvolvimento de uma técnica sistemática para resolver um desejado problema. De um modo geral, o interesse pelo algoritmo é inerente ao estudo do problema. Contudo, este conceito somente foi fundamentado nas primeiras décadas do século passado, através da formalização da noção da computação. Isto possibilitou demonstrar a existência de problemas algorítmicos que não podem ser resolvidos. Com isso, abriu-se um novo campo de interesses, que consistia em identificar e classificar os problemas, segundo a existência ou não de algoritmo para resolvê-los.



Figura 1.3: Um algoritmo

O aparecimento do computador veio influenciar enormemente o panorama do estudo de algoritmos. Antes deste evento, por exemplo, o problema de melhorar a eficiência de tempo de um algoritmo era, em geral, de importância marginal. O problema de aumentar a eficiência de espaço era ainda mais irrelevante. O aparecimento das máquinas tornou possível a implementação e computação automática dos algoritmos. Além disso, desenvolveu-se um enorme número de aplicações de interesse prático que depende de resultados reais obtidos através desses algoritmos. Como consequência natural, deixou de ser o bastante assinalar apenas a existência ou não de algum algoritmo para um dado problema. Tornou-se também necessário impor que o tempo e espaço consumidos pela máquina para a implementação do algoritmo estejam dentro dos limites da prática. Por exemplo, resolver o problema do caminho hamiltoniano segundo o método das permutações, descrito anteriormente, para um valor  $n = 20$ , é certamente muito além da realidade. Ou seja, do ponto de vista da aplicação é como se aquele algoritmo talvez não existisse.

### 1.3 Apresentação dos Algoritmos

Conforme mencionado, um algoritmo pode ser associado a uma estratégia para resolver um desejado problema. Os dados do problema constituem a *entrada* ou os *dados* do algoritmo. A solução do problema corresponde a sua *saída*. Um algoritmo poderia ser então caracterizado por uma função  $f$ , a qual associa uma saída  $S = f(E)$ , a cada entrada  $E$  (Figura 1.3). Diz-se então que o algoritmo *computa* a função  $f$ . Assim sendo, é justificável considerar a entrada como a variável independente básica, em relação à qual são produzidas as saídas do algoritmo, bem como são analisados os comportamentos de tempo e espaço do mesmo.

Os algoritmos neste texto serão apresentados em uma linguagem quase livre de formato, contendo porém sentenças especiais com significado pré-definido. Isto é, na linguagem de apresentação dos algoritmos, há flexibilidade suficiente para que se possa descrever livremente em língua portuguesa a estratégia desejada. Contudo, para tornar a apresentação mais simples e curta é possível também a utilização de um conjunto de sentenças especiais, às quais correspondem algumas operações que aparecem com grande frequência nos algoritmos. Para maior simplicidade de exposição, o termo algoritmo será utilizado também com o significado de apresentação do algoritmo.

A linguagem de apresentação assemelha-se a um tipo ALGOL em português. Utilizamos a nomenclatura ALGOL, por motivos históricos, pois esta foi a primeira linguagem de programação amplamente utilizada a empregar estruturação. Possui uma estrutura cujo objetivo seria o de facilitar a descrição e implementação dos processos. Sempre que desejarmos fazer referência a linguagens desse tipo, mencionamos o ALGOL, o qual deve ser interpretado com um significado mais amplo. Isto é, aquele definido e utilizado em diversas linguagens posteriores providas de estruturação. Esta estrutura deve ser sempre obedecida na apresentação, mesmo quando se utiliza o português livre de formato. As ideias gerais nas quais se baseia são as seguintes:

A apresentação de um algoritmo em uma folha de papel é naturalmente dividida em *linhas*. Cada linha possui uma *margem* correspondente ao ponto da folha de papel onde se inicia a linha em questão. Sejam  $r$ ,  $s$  duas linhas de um algoritmo,  $r$  antecedendo  $s$ . Diz-se que a linha  $r$  *contém* a linha  $s$  quando (i) a margem de  $r$  é mais à esquerda do que a de  $s$  e (ii) se  $t \neq r$ , é uma linha do algoritmo compreendida entre  $r$  e  $s$ , então a margem de  $r$  é também mais à esquerda que a de  $t$ . Por exemplo, a linha  $p$  contém a linha  $q$  na Figura 1.4(a), mas não a contém nas 1.4(b) e (c).

A unidade básica da estrutura de apresentação dos algoritmos é o bloco. Um *bloco* é um conjunto composto de uma linha qualquer  $r$  com todas as linhas  $s$  que  $r$  contém. Consequentemente as linhas que formam um bloco são

	p .....	calcular o grau $g(v)$
(a)	z .....	eliminar arestas paralelas de $G$
	q .....	seja $C$ uma componente conexa de $G$
	p .....	calcular o grau $g(v)$
(b)	z .....	eliminar arestas paralelas de $G$
	q .....	seja $C$ uma componente conexa de $G$
	p .....	calcular o grau $g(v)$
(c)	z .....	eliminar arestas paralelas de $G$
	q .....	seja $C$ uma componente conexa de $G$

Figura 1.4: Linhas de um algoritmo

necessariamente contíguas. Além disso, um bloco é sempre formado por uma sequência de blocos contíguos. Isto é, um bloco  $B$  pode ser decomposto como  $B = B_1 \cup B_2 \cup \dots \cup B_k$  onde cada  $B_i$  é um bloco, sendo  $B_j$  contíguo a  $B_{j+1}$ ,  $1 \leq j < k$ .

Observe que a margem esquerda de uma linha atua como delimitador de blocos, cumprindo papel semelhante ao desempenhado pelos *begin's* e *end's* do ALGOL. No exemplo da Figura 1.4(a), o bloco iniciado pela linha  $p$  contém  $z$  e  $q$ . Na Figura 1.4(b), ele contém apenas  $p$ , mas o iniciado por  $z$  contém  $q$ . Na Figura 1.4(c), os blocos com início em  $p$  e  $z$  são ambos formados por uma única linha.

Dentre os blocos a que uma linha  $r$  pertence, um possui especial interesse. É o uma *bloco* definido por  $r$ , que é o maior de todos que se iniciam na linha  $r$ .

Além da estrutura descrita anteriormente, a linguagem de apresentação dos algoritmos possui também sentenças especiais. A utilização ou não dessas sentenças é opcional. Contudo, se empregadas devem obedecer à estrutura geral acima. Além disso, cada uma delas possui uma sintaxe própria. Uma sentença especial inicia uma linha da apresentação e, portanto, inicia um bloco chamado *bloco especial*. Este, naturalmente, corresponde à porção do algoritmo onde se estende o efeito da sentença correspondente.

As seguintes sentenças especiais são de interesse.

(i) **algoritmo** *<comentário>*

Se presente, esta linha deve ser a primeira da apresentação do algoritmo e o bloco definido por **algoritmo** deve conter toda a apresentação. O *<comentário>* é opcional e normalmente se refere à finalidade do **algoritmo**.

(ii) **dados:** *<Descrição>*

Esta sentença é utilizada para informar os dados de entrada do algoritmo. A *<Descrição>* deve caracterizar esses dados, em formato livre. Normalmente, a única sentença que antecede **dados** é a (i) acima.

(iii) **procedimento** *<nome>*

Um **procedimento** é equivalente ao do definido em ALGOL. Assim sendo, ele altera a ordem sequencial de interpretação dos algoritmos. O bloco definido por **procedimento** deve ser saltado na computação sequencial do processo. A interpretação desse bloco é realizada imediatamente após a detecção de uma linha que invoque o *<nome>* do **procedimento**. Encerrado este último, a sequência de interpretação retorna para logo após o ponto interrompido. O *<nome>* pode envolver parâmetros, como em ALGOL.

(iv) **se** *<condição>* **então** *<sentença 1>* **caso contrário** *<sentença 2>*

É semelhante ao **if ... then ... else ...** do ALGOL. Se a *<condição 1>* é verdadeira então *<sentença 1>* é interpretada e *<sentença 2>* desconsiderada. Se a *<condição>* for falsa, vale o oposto. Em qualquer caso, o bloco seguinte a esta sentença é interpretado após. A parte **caso contrário** *<sentença 2>* pode ser omitida.

(v) **para** *<variação>* **efetuar** *<sentença>*

É semelhante ao **for ... do ...** do ALGOL. A *<variação>* consiste em um conjunto  $S = \{s_1, \dots, s_k\}$  e de uma variável  $j$ , sendo denotada por  $s_j \in S$ . Nesse caso, o bloco definido por esta sentença é interpretado  $k$

vezes, a primeira com  $j = s_1$ , a segunda com  $j = s_2$ , e assim por diante. A *<variação>* pode também ser denotada por  $j = s_1, s_2, \dots, s_k$ .

(vi) **enquanto** *<condição>* **efetuar** *<sentença>*

Semelhante ao **while ... do ...** do ALGOL. A *<condição>* é avaliada e se for verdadeira, o bloco definido por esta sentença é interpretado. Após o qual a *<condição>* é novamente avaliada e o processo se repete. Se a *<condição>* é falsa, então o bloco em questão deve ser saltado.

(vii) **repetir**

...

...

**até que** *<condição>*

Semelhante ao **repeat ... until** do ALGOL. O bloco subsequente à linha que contém **repetir** é interpretado. Ao atingir a linha **até que** a condição é avaliada. Caso seja falsa, todo o processo se repete. Caso contrário, a sequência de interpretação passa a linha seguinte a que contém **até que**.

As demais sentenças podem ser escritas em formato livre, atendendo contudo às condições da estrutura em blocos. Utiliza-se, obviamente, símbolos matemáticos correntes e o *operador de atribuição* := conforme em ALGOL (isto é,  $x := y$  significa atribuir a  $x$  o valor de  $y$ , permanecendo este último com o seu valor antigo).

Como exemplo da utilização da linguagem de apresentação de algoritmos, considere o seguinte caso. É dada uma sequência  $s(1), s(2), \dots, s(n)$  de termos. O objetivo é inverter a sua ordem, isto é, rearranjar os termos na sequência, de modo que  $s(n)$  apareça na posição 1,  $s(n - 1)$  na posição 2, e assim por diante. O Algoritmo 1.1 descreve o processo.

---

**Algoritmo 1.1:** Inversão de uma sequência

---

**Dados:** sequência  $s_1, s_2, \dots, s_n$

**para**  $j = 1, 2, \dots, \lfloor n/2 \rfloor$  **efetuar**

$b := s(j)$

$s(j) := s(n - j + 1)$

$s(n - j + 1) := b$

---

## 1.4 Complexidade de Algoritmos

Conforme já mencionado, na área de algoritmos em grafos (bem como algoritmos, de um modo geral) é inerente uma preocupação computacional para com os processos desenvolvidos. Essa preocupação, por sua vez, implica o objetivo de procurar elaborar algoritmos que sejam os mais eficientes possíveis. Consequentemente, torna-se imperioso o estabelecimento de critérios que possam avaliar esta eficiência.

De início, os critérios de medida de eficiência eram em geral empíricos, principalmente no que se refere à eficiência de tempo. Baseado em uma certa estratégia, um algoritmo era descrito e implementado. Em seguida, efetuava-se uma avaliação prática de seu comportamento. Isto é, um programa implementando o algoritmo era executado em um computador, para alguns conjuntos de dados de entrada diferentes. Para cada execução media-se o tempo correspondente. Ao final da experiência eram obtidas algumas curvas destinadas a avaliar o comportamento de tempo do algoritmo.

Em geral, esses eram os critérios utilizados até aproximadamente a primeira metade do século passado. Embora as informações obtidas através desse processo sejam também úteis, os critérios mencionados não permitem aferir, com exatidão, o comportamento do algoritmo. E por vários motivos. As curvas obtidas traduzem apenas os resultados de medidas empíricas para dados particulares. Além disso, essas medidas são dependentes de uma implementação particular (portanto, da qualidade de programador), do compilador específico utilizado, do computador empregado e até das condições locais de processamento no instante da realização das medidas.

Estes fatos justificam a necessidade da adoção de algum processo analítico para avaliação da eficiência. O critério descrito a seguir tenta se aproximar deste objetivo.

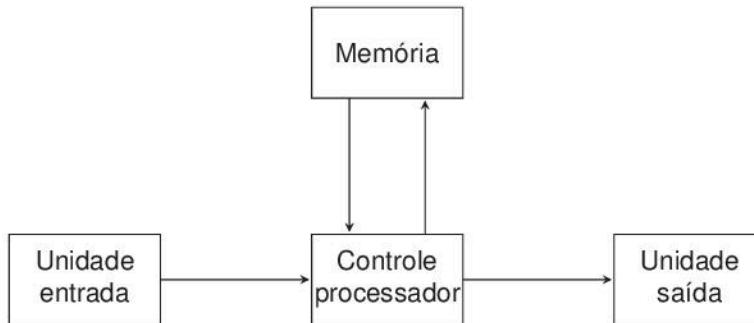


Figura 1.5: Um modelo computacional

A tarefa de definir um critério analítico torna-se mais simples se a eficiência a ser avaliada for relativa a alguma máquina específica. Recorde que o método de avaliação empírico mencionado anteriormente, produz sempre resultados relativos ao computador utilizado nas experiências. Ao invés de escolher uma máquina particular, em relação a qual a eficiência dos algoritmos seria avaliada, é certamente mais conveniente utilizar-se um modelo matemático de um computador. Uma possível formulação desse modelo é a RAM (random access machine). Este é composto de uma *unidade de entrada*, *unidade de saída*, *memória* e *controle/processador* (Figura 1.5).

O funcionamento de uma RAM é semelhante ao de um computador hipotético elementar. O processador dispõe de *instruções* que podem ser executadas. A memória armazena os *dados* e o *programa*. Este último consiste em um conjunto de instruções que implementa o algoritmo. Cada instrução  $I$  do modelo possui um *tempo de instrução*  $t(I)$ . Assim sendo, se para a execução de um programa  $P$ , para uma certa entrada fixa, são processadas  $r_1$  instruções do tipo  $I_1$ ,  $r_2$  instruções do tipo  $I_2, \dots, r_m$  instruções do tipo  $I_m$ , então o *tempo de execução* do programa  $P$  é dado por  $\sum_{j=1}^m r_j \cdot t(I_j)$ . O que se segue é uma tentativa de avaliação deste somatório.

Introduz-se a seguinte simplificação adicional. Supõe-se que  $t(I) = 1$  para toda instrução  $I$ , isto é, que o tempo de execução de cada instrução seja constante e igual a 1. À primeira vista, essa simplificação talvez se afigure como não razoável. Para justificá-la, observe inicialmente que a relação  $t(I_1)/t(I_2)$  entre os tempos de execução das instruções  $I_1$  e  $I_2$  pode ser aproximada a uma constante, considerando-se fixos os tamanhos dos operandos. Por exemplo, uma instrução de multiplicação seria aproximadamente  $k$  (constante) vezes mais lenta que uma comparação. Essa simplificação parece razoável. Sejam agora  $T'$  e  $T''$  respectivamente, os valores dos tempos de execução do programa, para os caso em que  $t(I_1)/t(I_2) = \text{constante}$  e  $t(I) = 1$ . Isto é,  $T'$  corresponde ao valor do tempo de execução do programa, supondo apenas a simplificação de que a relação entre dois tempos de instruções arbitrárias é uma constante. Enquanto isso,  $T''$  corresponde ao tempo de execução do programa no caso em que os tempos de instruções são todos unitários. Segue-se então que existe uma constante que limita  $T'/T''$ , ou seja, a simplificação adicional  $t(I) = 1$  introduz uma distorção de exatamente uma constante, em relação à simplificação anterior  $t(I_1)/t(I_2) = k$ .

Com  $t(I) = 1$ , o valor do tempo de execução de um programa torna-se igual ao número total de instruções computadas. Denomina-se *passo de um algoritmo*  $\alpha$  à computação de uma instrução do programa  $P$  que o implementa. A *complexidade local* do algoritmo  $\alpha$  é o número total de passos necessários para a computação completa de  $P$ , para uma certa entrada  $E$ . Assim sendo, considerando-se as simplificações introduzidas, a complexidade local de  $\alpha$  é equivalente ao seu tempo de execução, para a entrada  $E$ . O interesse é determinar o número total de passos acima, para entradas consideradas suficientemente grandes.

Nesse sentido, torna-se relevante avaliar o tamanho da entrada. Supondo que esta seja composta de  $n$  símbolos, o seu comprimento seria o somatório dos tamanhos das codificações correspondentes aos  $n$  símbolos, segundo o critério de codificação utilizado. Para simplificar a análise, a menos que seja explicitamente dito em contrário, o tamanho da codificação de cada símbolo será considerado constante. Assim sendo, o tamanho da entrada pode ser expresso por um número proporcional a  $n$ , sendo considerado grande quando  $n$  o for. Esta simplificação é amplamente satisfatória quando a codificação de cada símbolo puder ser armazenada em uma palavra de computador,

como ocorre frequentemente. Nesse caso, o número de palavras utilizadas exprime obviamente o comprimento da entrada.

A avaliação da eficiência para problemas grandes é de certa forma a mais importante. Além disso, facilita a manipulação de sua expressão analítica. Consequentemente, seria também razoável aceitar uma medida analítica que refletisse um limite superior para o número de passos, em lugar de um cálculo exato. Define-se então *complexidade (local) assintótica* de um algoritmo como sendo um limite superior da sua complexidade local, para uma certa entrada suficientemente grande. Naturalmente, em se tratando de um limite superior, o interesse é determiná-lo o mais justo, ou seja, o menor possível. A complexidade assintótica deve ser escrita em relação às variáveis que descrevem o tamanho da entrada do algoritmo.

Para exprimir analiticamente a complexidade assintótica é conveniente utilizar a notação seguinte, denominada *notação O*. Seja  $f$  uma função real não negativa da variável inteira  $n \geq 0$ . Diz-se que  $f$  é  $O(h)$ , denotando-se  $f = O(h)$ , quando existirem constantes  $c, n_0 > 0$  tais que  $f(n) \leq ch$ , para  $n \geq n_0$ . Por exemplo,  $3x^2 + 2x + 5$  é  $O(x^2)$ ,  $3x^2 + 2x + 5$  é  $O(x^3)$ ,  $4x^2 + 2y + 5x$  é  $O(x^2 + y)$ ,  $2x + \log x + 1$  é  $O(x)$ , 542 é  $O(1)$ , e assim por diante. É imediato verificar que  $O(h_1 + h_2) = O(h_1) + O(h_2)$  e  $O(h_1 \cdot h_2) = O(h_1) \cdot O(h_2)$ . Além disso, se  $h_1 \geq h_2$ , então  $O(h_1 + h_2) = O(h_1)$ . Seja  $k$  uma constante, então  $O(k \cdot h) = k \cdot O(h) = O(h)$ . Observe que termos de ordem mais baixa são irrelevantes. Quando a função  $f$  é  $O(h)$  diz-se também que  $f$  é da *ordem* de  $h$ .

A complexidade assintótica de um algoritmo obviamente não é única, pois a entradas diferentes podem corresponder números de passos diferentes. Dentro dessa diversidade de entradas, é sem dúvida importante aquela que corresponde ao *pior caso*. Talvez para a maioria das aplicações esse é o caso mais relevante. Além disso, é de tratamento analítico mais simples. Define-se então *complexidade de pior caso* (ou simplesmente *complexidade*) de um algoritmo como o valor máximo dentre todas as suas complexidades assintóticas, para entradas de tamanho suficientemente grandes. Ou seja, a complexidade de pior caso traduz um limite superior do número de passos necessários à computação da entrada mais desfavorável, de tamanho suficientemente grande.

A complexidade de um algoritmo é sem dúvida um indicador importante para a avaliação da sua eficiência de tempo. Mas certamente também possui aspectos desvantajosos e tampouco é o único indicador existente. A complexidade procura traduzir analiticamente uma expressão da eficiência de tempo do pior caso. Contudo, há exemplos em que é por demais pessimista. Isto é, o pior caso pode corresponder a um número de passos bastante maior do que os casos mais frequentes. Além disso, a expressão da complexidade não considera as constantes. Este é um outro fator onde distorções podem ser introduzidas. Seja o exemplo em que o número de passos necessários para a computação do pior caso de um algoritmo é  $c_1x^2 + c_2x$  onde  $c_1, c_2$  são constantes com  $c_1 \ll c_2$  e  $x$  uma variável que descreve o tamanho da entrada. Então porque a complexidade é assintótica ela seria escrita como  $O(x^2)$ , o que pode não exprimir satisfatoriamente as condições do exemplo.

Por analogia, define-se também um indicador para o melhor caso do algoritmo. Assim sendo, a *complexidade de melhor caso* é o valor mínimo dentre todas as complexidades assintóticas do algoritmo, para entradas suficientemente grandes. Isto é, a complexidade de melhor caso corresponde a um limite inferior do número de passos necessários à computação da entrada mais favorável, de tamanho suficientemente grande. Analogamente ao pior caso, este novo indicador deve ser expresso em função do tamanho da entrada.

A notação  $\Omega$  seguinte é útil no estudo de complexidade. Seja  $f$  uma função real não negativa da variável inteira  $n > 0$ . Então  $f = \Omega(h)$  significa que existem constantes  $c, n_0 > 0$ , tais que  $f(n) \geq ch$ , para  $n \geq n_0$ . Por exemplo,  $5n^2 + 2n + 3$  é  $\Omega(n^2)$ . Também  $2n^3 + 5n$  é  $\Omega(n^2)$ , e assim por diante.

Seja  $P$  um problema algorítmico, cuja entrada possui tamanho  $n > 0$ , e  $g$  uma função real positiva da variável  $n$ . Diz-se que  $\Omega(g)$  é um *limite inferior* de  $P$  quando qualquer algoritmo  $\alpha$  que resolva  $P$  requerer pelo menos  $O(g)$  passos. Isto é, se  $O(f)$  for a complexidade (pior caso) de  $\alpha$  então  $g$  é  $O(f)$ . Por exemplo, se  $\Omega(n^2)$  for um limite inferior para  $P$ , não poderá existir algoritmo que resolva  $P$  em  $O(n)$  passos. Um algoritmo  $\alpha_0$  que possua complexidade  $O(g)$  é denominado *ótimo* e, nesse caso,  $g$  é o *limite inferior máximo* de  $P$ . Observe que  $\alpha_0$  é o algoritmo de complexidade mais baixa dentre todos os que resolvem  $P$ . Este novo indicador é obviamente importante e, frequentemente, de difícil determinação. É também mais geral que os anteriores, pois é relativo ao problema e não a alguma solução específica.

Como exemplo, seja o Algoritmo 1.1 da Seção 1.3 para inversão da ordem dos termos de uma sequência. A sua entrada consiste na sequência  $s_1, \dots, s_n$ , onde cada  $s_i$  possui tamanho constante. Ou seja, a entrada possui tamanho  $O(n)$ . O procedimento corresponde a um bloco composto por três operações de atribuição, o qual deve ser computado  $n/2$  vezes. Cada uma dessas operações pode ser computada em tempo constante. Logo o número total de passos é  $O(n)$ . Observe que esse algoritmo, em particular, não é sensível à entrada. Isto é, qualquer que seja a entrada, o algoritmo efetua  $O(n)$  passos. Então, naturalmente, as complexidades de pior e melhor casos são ambas iguais a  $O(n)$ .

Recorde que a complexidade de um algoritmo é um indicador relativo ao modelo matemático RAM. Uma questão natural seria saber se a sua expressão pode ser utilizada para avaliar também o comportamento desse algoritmo, quando implementado em uma máquina real. A resposta é, felizmente, sim. Para ilustrar um caso, considere novamente o Algoritmo 1.1. Cada operação de atribuição mencionada em geral corresponde a  $m$  (constante  $> 1$ ) instruções na máquina real. Portanto, é constante a relação entre os tempos de execução, do Algoritmo 1.1, respectivamente quando implementado em duas máquinas diferentes. Ou seja, a complexidade  $O(n)$  independe da implementação particular realizada (supondo, obviamente, uma implementação adequada, para cada caso). Esta observação geral justifica também a simplificação correspondente de adotar como critério de avaliação de eficiência expressões de complexidade determinadas a menos de constantes.

Como exemplo adicional, considere o *problema de ordenação*. Dada uma sequência  $S$  de números, o objetivo consiste em dispô-los em ordem não decrescente. Sejam  $s_i, s_j \in S$ . Se  $s_i < s_j$  e  $s_i$  sucede  $s_j$  em  $S$ , então diz-se que o par  $s_i, s_j$  forma uma *inversão*. Por exemplo, a sequência 2 5 3 4 apresenta duas inversões 5 3 e 5 4. A sequência estará ordenada exatamente quando não apresentar inversões. Seja  $s_i, s_j$  uma inversão. Se  $s_i, s_j$  intercambiarem posições em  $S$ , então  $s_i, s_j$  obviamente deixa de ser inversão. Esta troca de posições pode também criar ou eliminar outras inversões. Contudo, o número total das inversões eliminadas é maior do que o das criadas. Essas observações conduzem ao seguinte algoritmo de ordenação.

---

**Algoritmo 1.2:** Ordenação de uma sequência (básico)

---

**Dados:** sequência  $S = s_1, \dots, s_n$

**enquanto** existir uma inversão  $s_i, s_j$  **efetuar**  
trocar de posição  $s_i$  com  $s_j$

---

Para determinar a complexidade desse algoritmo, observe que o mesmo apresenta basicamente duas operações: (i) a identificação de uma inversão  $s_i, s_j$  e (ii) a correspondente troca de posições de  $s_i$  com  $s_j$ . Seja  $I$  o número total de inversões de  $S$ . A operação (ii) pode ser realizada obviamente em tempo constante, logo requer  $O(I)$  passos no total. A operação (i) pode ser realizada, sem dificuldade, em  $O(n)$  passos por inversão. Para tal, basta percorrer  $S$  da esquerda para a direita, comparando cada número da sequência com o seu antecedente. Portanto, para identificar todas as inversões do processo o Algoritmo 1.2 requer tempo  $O(nI)$ . A leitura dos dados é realizada em tempo  $O(n)$ . Logo, em uma primeira análise superficial, sua complexidade assintótica é  $O(nI + n)$ . As complexidades de pior e melhor caso podem ser calculadas através da atribuição de valores específicos a  $I$ . O valor máximo do número de inversões  $I$  corresponde ao caso em que a sequência de entrada  $S$  se encontra em ordem decrescente. Isto é, qualquer par de elementos forma inversão. Portanto,  $I_{\max} = n(n - 1)/2$ .

O valor mínimo de  $I$  ocorre quando  $S$  já se encontra ordenada. Isto é,  $I_{\min} = 0$ . Logo, as complexidades de pior e melhor caso são  $O(n^3)$  e  $O(n)$ , respectivamente.

Para tornar o processo mais eficiente pode-se adotar um critério segundo o qual as subsequências de  $s_1$  até  $s_j$  são mantidas ordenadas, para valores crescentes de  $j$ . Isto é, no passo inicial, o problema é restrito à subsequência  $S(1) = s_1$ , já ordenada trivialmente. No passo geral, supõe-se que esteja ordenada a subsequência  $S(j - 1)$  formada pelos  $j - 1$  elementos iniciais de  $S$ . Acrescente  $s_j$  à direita em  $S(j - 1)$ . A ideia consiste em sucessivamente comparar  $s_j$  de  $S$  com o elemento de  $S(j - 1)$  imediatamente à sua esquerda. Se esse par formar inversão efetua-se a correspondente troca de posições, e assim por diante. Caso contrário, ou se  $s_j$  se encontrar na primeira posição da subsequência, então  $S(j)$  estará também ordenada. O Algoritmo 1.3 descreve o processo.

**Algoritmo 1.3:** Ordenação de uma sequência

---

**Dados:** sequência  $S = s_1, \dots, s_n$

**para**  $j = 1, \dots, n - 1$  **efetuar**

$k := j$

**enquanto**  $s_k > s_{k+1}$  e  $k \geq 1$  **efetuar**

trocar de posição  $s_k$  com  $s_{k+1}$

$k := k - 1$

---

O efeito principal produzido por esta alteração é tornar constante o número de operações necessárias para identificar as novas inversões criadas pela adição do elemento  $s_j$ . Assim sendo, a complexidade da ordenação decresce para  $O(n + I)$ . Isto é, a complexidade de pior do algoritmo acima é  $O(n^2)$ .

Observamos que a complexidade  $O(n^2)$  para realizar a ordenação de  $n$  elementos pode ainda ser reduzida a  $O(n \log n)$ . No Capítulo 3 será demonstrado que esta última é ótima para algoritmos que resolvem o problema de ordenação através de comparação de seus elementos.

Foi discutido nesta seção o problema de conceituar um critério de avaliação de eficiência de algoritmos. Apesar de ter sido mencionado também o espaço, somente a eficiência de tempo foi considerada. A análise do espaço requerido por um algoritmo é um problema, em geral, mais simples do que o estudo do tempo. De forma análoga, podem ser definidas para o espaço a complexidade assintótica, a de pior caso (ou simplesmente complexidade) e a de melhor caso. Frequentemente, estas podem ser obtidas sem maiores dificuldades através de expressões na notação  $O$ . Por exemplo, os algoritmos de ordenação descritos anteriormente possuem complexidades de espaço de pior e melhor caso, ambas iguais a  $O(n)$ .

## 1.5 Estruturas de Dados

Nesta seção são apresentadas, de forma sumária, algumas estruturas de dados de interesse ao texto. Supõe-se que o leitor já esteja familiarizado com o assunto, principalmente com os algoritmos de manipulação dessas estruturas. O objetivo desta apresentação é apenas descrever a nomenclatura utilizada.

Uma *lista* é simplesmente uma sequência de elementos. Utiliza-se a notação  $L = \{a_1, \dots, a_n\}$ , onde os  $a_i$  são os elementos da lista. Uma maneira simples de representar  $L$  em um esquema de memória de computador consiste em alocar seus elementos  $a_1, \dots, a_n$  sequencialmente na memória. Neste caso,  $L(j)$  denota o elemento  $a_j$ ,  $1 \leq j \leq n$ , sendo portanto determinado pelo índice  $j$ . Esta forma de alocação denomina-se *representação sequencial* de  $L$ , enquanto a lista  $L$  recebe o nome de *lista sequencial* ou *vetor*.

Um outro modo de representar  $L$  consiste em alocar a lista de elementos  $a_1, \dots, a_n$ , respectivamente, em posições quaisquer da memória. Nesse caso, a fim de que a lista possa ser manipulada há necessidade de associar uma variável especial  $t_j$  chamada *ponteiro* a cada elemento  $a_j$  da lista. Para  $j < n$ , o ponteiro  $t_j$  informa a localização do elemento  $a_{j+1}$ .

Define-se também  $t_n = \emptyset$ , o que indica o final da lista. Uma variável especial  $t_0$ , chamada *ponteiro inicial*, informa a localização do primeiro elemento da lista. Este esquema de alocação é denominado *representação por ponteiros*, sendo  $L$  chamada *lista encadeada*. Nesta representação, em geral, o par  $a_j, t_j$  é alocado sequencialmente na memória, pois a localização de  $t_j$  deve ser obtida implicitamente a partir de  $a_j$ .

Uma lista que não possui elementos é chamada *vazia*. Uma forma de representar uma lista vazia consiste em definir um ponteiro inicial  $t_0$ , com  $t_0 = \emptyset$ .

Uma lista  $L = \{a_1, \dots, a_n\}$ , na representação por ponteiros é tal que o ponteiro  $t_n$  informa a localização do elemento  $a_1$  (ao invés de  $t_n = \emptyset$ ), recebe o nome de *lista circular*. As Figuras 1.6(a), (b) e (c) ilustram uma lista  $L$  com elementos  $\{a, b, c, d, e\}$ , nos casos, respectivamente, em que  $L$  é um vetor, uma lista encadeada e uma lista circular. A notação gráfica  $\perp$  corresponde ao símbolo  $\emptyset$ .

Há aplicações em que é conveniente representar uma lista  $L = \{a_1, \dots, a_n\}$  da seguinte maneira alternativa. Seus elementos estão em posições quaisquer da memória. A cada elemento  $a_j \in L$  é associado um par  $s_j, t_j$  de *ponteiros*. O valor  $s_j$  indica a posição de  $a_j - 1$ , enquanto  $t_j$  informa a de  $a_{j+1}$ . Define-se  $s_1 = t_n = \emptyset$ . Da

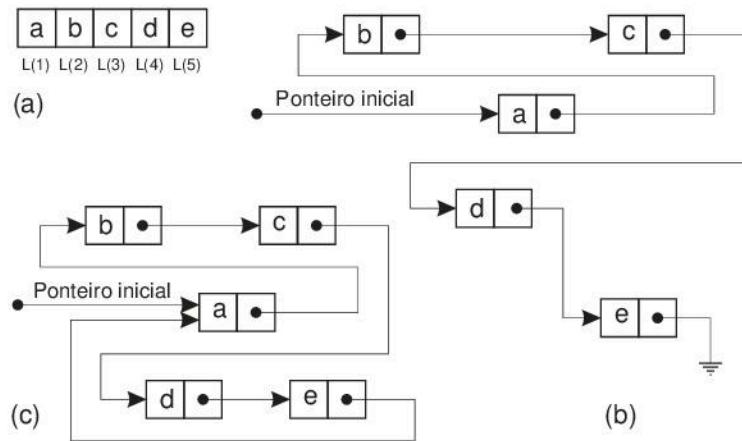


Figura 1.6: Representações diferentes de uma lista

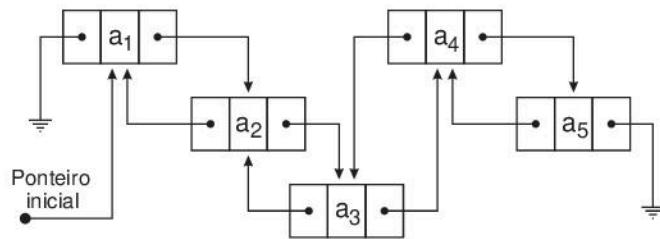


Figura 1.7: Uma lista duplamente encadeada

mesma forma, uma variável especial  $t_0$ , o *ponteiro inicial*, indica a localização de  $a_1$ . Este esquema é denominado *representação por duplos ponteiros da lista*, sendo esta denominada *lista duplamente encadeada*. Ver Figura 1.7. Quando os ponteiros  $s_1$  e  $t_n$  informam, respectivamente, as localizações de  $a_n$  e  $a_1$ , então  $L$  denomina-se *lista circular duplamente encadeada*.

As modalidades de listas apresentadas até o momento diferem entre si apenas no que toca à representação. Isto é, não foram estabelecidas restrições quanto ao tipo de informação de seus elementos, nem quanto à forma de manipulação das listas. Os tipos de listas descritas mais adiante diferem quanto a este último aspecto.

As operações básicas efetuadas em uma lista  $L$  são as de *inclusão* e *exclusão* de um elemento. Elas correspondem respectivamente à inserção de um novo elemento em  $L$  e à retirada de algum outro da lista. Os algoritmos de inclusão e exclusão de elementos em uma lista são simples e não serão apresentados neste texto.

Os dois tipos de listas seguintes são bastante frequentes. Uma *pilha* é uma lista em que todo elemento a ser excluído é necessariamente o último incluído. A ideia de uma pilha pode ser visualizada imaginando-se um conjunto de pratos empilhados. Nessa disposição, a introdução e retirada de um prato se dá somente pelo topo. Assim sendo, todo prato a ser excluído é o último que foi incluído. Uma *fila* é uma lista em que todo elemento a ser excluído é necessariamente o primeiro incluído, presente na lista. Ou seja, numa pilha se exclui apenas o elemento mais novo, enquanto que numa fila é excluído o mais antigo. Uma fila pode ser visualizada, imaginando-se simplesmente uma fila de pessoas. Nesta, toda pessoa a ser servida deve ser a mais antiga da fila. As Figuras 1.8(a) e (b) ilustram, respectivamente, esquemas de pilha e fila.

Observe que uma lista  $L$  mesmo quando encadeada é uma estrutura necessariamente sequencial. Assim sendo, quando  $L$  é não vazia é sempre possível identificar o primeiro e o último elementos da sequência, os quais são denominados *extremidades* de  $L$ . Em particular, as operações de inclusão e exclusão nas listas especiais anteriores são efetuadas apenas nas suas extremidades, conforme indica a Figura 1.8. Numa pilha, essas operações são ambas realizadas obrigatoriamente em uma só extremidade, chamada *tipo da pilha*. Em uma fila, as exclusões são realizadas todas em uma mesma extremidade, enquanto que as inclusões são efetuadas na outra. Essas extremidades são denominadas *início* e *final* da fila, respectivamente.

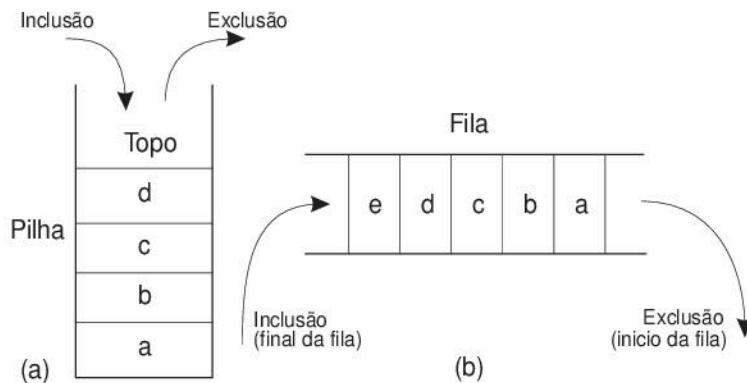


Figura 1.8: Pilha e fila

Esses fatos conduzem à seguinte generalização. Uma *fila dupla* é uma lista  $L$  em que as inclusões e exclusões são efetuadas apenas nas extremidades. Isto é, se  $L$  possui mais de um elemento há exatamente duas localizações possíveis, as extremidades, para se inserir ou retirar elementos de  $L$ . Quando as exclusões forem restritas a uma só extremidade,  $L$  denomina-se *fila dupla de saída restrita*, enquanto se as inclusões forem restritas a estrutura é uma *fila dupla de entrada restrita*. Finalmente, menciona-se que a inclusão ou exclusão de um elemento em uma fila dupla pode ser efetuada em um número constante de passos, através de algoritmo simples.

## 1.6 A Linguagem Python

Como já mencionado, os algoritmos serão apresentados em uma linguagem quase livre de formato, o que é conveniente em muitos aspectos. Entre eles, em especial, a separação entre a lógica da resolução do problema, e os requerimentos e limitações práticas que o uso de uma determinada linguagem impõe. Como exemplos de tais considerações que devem ser feitas após a elaboração do algoritmo, podemos citar a importação de bibliotecas, adequação a uma gramática mais rígida (como o uso de delimitadores de fim de linha e de início e fim de blocos que a gramática de algumas linguagens requerem), a transformação para uma notação mais distante da linguagem matemática utilizada no algoritmo, entre outras. Desta forma, assim que um algoritmo é discutido e o método de solução é compreendido, pode surgir um obstáculo para o iniciante no desenvolvimento de programas: como transpor este algoritmo para uma linguagem de programação executável? Este passo não pode ser desprezado, pois a execução de experimentos reais com os algoritmos é crucial no processo de aprendizado. Na tentativa de ajudar a transpor esta barreira, apresentamos ao final de cada capítulo uma seção que ilustra a implementação de cada algoritmo na linguagem de programação Python (na sintaxe de sua Versão 3). No restante desta seção, trataremos de descrever as estruturas básicas que foram apresentadas para os algoritmos e mostrar como elas serão transformadas para a linguagem Python.

Em Python, os blocos são definidos de forma análoga àquela dos algoritmos, com relação às margens de cada linha. No caso do uso de um editor de texto, as margens são chamadas de indentações.

As sentenças especiais utilizadas nos algoritmos possuem em geral uma estrutura equivalente na linguagem de programação. No caso da transformação para Python, mapearemos as sentenças dos algoritmos, descritos na seção anterior, em geral, da seguinte maneira.

### (i) **algoritmo <comentário>**

Esta sentença é utilizada opcionalmente para elaborar comentários especiais sobre o algoritmo e, assim sendo, transforma-se naturalmente em comentários na linguagem Python, em suas duas formas possíveis:

1    `#algoritmo <comentário>`

ou

1    `"""algoritmo <comentário>`

```

2   muito
3   longo>"""

```

(ii) **dados:** <descrição>

Transformado em comentário analogamente à transformação de “algoritmo” no item anterior.

(iii) **procedimento** <nome>

Em Python, a palavra reservada `def` é utilizada para indicar um procedimento e, os dados de entrada descritos no item anterior, são formalmente definidos como argumentos (parâmetros). Assim, uma transformação típica de um procedimento seria da seguinte forma:

```

1 #dados: Grafo G e um vértice v de G
2 def EscreverComponenteConexa(G, v):
3     ...

```

(iv) **se** <condição> **então** <sentença 1> **caso contrário se** <condição 2> **então** <sentença 2> **caso contrário** <sentença 3>

Transforma-se para a estrutura abaixo. Note que o uso de “caso contrário se” em algoritmos possui uma palavra própria em Python.

```

1 if <condição 1>:
2     <sentença 1>
3 elif <condição 2>:
4     <sentença 2>
5 else:
6     <sentença 3>

```

(v) **para** <variação> **efetuar** <sentença>

A transformação depende da natureza de <variação>. Se <variação> se refere a uma variável (digamos, `i`) que assume valores de uma sequência cujo primeiro valor é `ini` e o último não é igual nem ultrapassa o valor `fim`, em incrementos de valor `passo` a partir do valor inicial, a transformação será feita por

```

1 for i in range(ini,fim,passo):
2     <sentença>

```

Quando `passo` é omitido, ele é assumido ser unitário. Se, por outro lado, <variação> for uma variável (digamos `x`) que assume cada valor de um conjunto  $X$  (o que significa dizer que, em Python, a estrutura que representa  $X$  é iterável, como é o caso de listas, por exemplo), a transformação é feita por

```

1 for x in X:
2     <sentença>

```

(vi) **enquanto** <condição> **efetuar** <sentença>

Transforma-se para a estrutura seguinte.

```

1 while <condição>:
2     <sentença>

```

(vii) **repetir** <sentença> **até que** <condição>

Em Python, não existe uma estrutura específica para esta forma de repetição. A equivalente, neste caso, é dada pela transformação abaixo (o comando `break` faz com que o fluxo de controle saia da repetição).

```

1 while True:
2     <sentença>
3     if <condição>:
4         break

```

O operador de atribuição em Python é o `=`, que funciona também com atribuição múltipla. Assim, a atribuição “`x := 10; y := 20`”, poderia ser expressa em Python da forma abaixo

```
1 x = 10; y = 20
```

ou

```
1 x, y = 10, 20
```

com resultados equivalentes. No entanto, nota-se que a atribuição múltipla não é equivalente a uma sequência de atribuições simples, mas uma atribuição na qual as expressões são atribuídas às respectivas variáveis de maneira atômica. Por exemplo, a troca de valores entre as variáveis `x` e `y` poderia ser feita da seguinte forma:

```
1 x, y = y, x
```

A representação dos vetores nos algoritmos é feita com listas de Python. As listas de Python necessariamente fazem corresponder o primeiro elemento à posição 0. Em geral, os vetores utilizados nos diversos algoritmos usam a convenção de que o primeiro elemento se encontra na posição 1. Assim, há duas formas usuais de conduzir a transformação. A primeira maneira, é fazer com que todo acesso à uma lista sempre busque por uma posição uma unidade a menos que a posição definida pelo algoritmo. A segunda maneira, que é a que devemos adotar, é que as listas em Python possuirão sempre um elemento a mais em relação ao vetor correspondente do algoritmo e de tal forma que o elemento de posição 0 da lista é atribuído a um valor conveniente qualquer e não é utilizado no decorrer do algoritmo. O espaço extra constante de um elemento é irrelevante perto da vantagem da implementação possuir menor diferença com seu algoritmo, facilitando o processo de conversão de algoritmo para a linguagem.

Como exemplo de transformação de um algoritmo inteiro, ilustramos aquela referente ao Algoritmo 1.1.

Programa 1.1: Inversão de uma sequência

```
1 #Algoritmo 1.1: Inversão de uma sequência
2 #Dados: sequência s[1],...,s[n]
3 def InversaoSequencia(s):
4     n = len(s)-1 #primeiro elemento é desconsiderado
5     for j in range(1,n//2+1):
6         s[j], s[n-j+1] = s[n-j+1], s[j]
```

Em alguns algoritmos, é utilizado como um valor numérico o valor “ $\infty$ ” (um abuso de notação, dado que qualquer valor numérico é definitivamente finito). Esta notação é útil para representar um valor numérico que é arbitrariamente grande, maior do que qualquer outro valor numérico que o algoritmo possa vir a processar. Em Python, há um valor com idêntico propósito, a saber, o `float("inf")`.

As listas de Python serão utilizadas também para representar pilhas. A próxima implementação ilustra a definição de uma pilha, seguido de duas operações de empilhamento e uma de desempilhamento.

```
1 Q = [] #define uma pilha (lista)
2 Q.append(3) #empilha o valor 3
3 Q.append(5) #empilha o valor 5
4 Q.pop() #desempilha o topo (valor 5)
```

Quanto à implementação de filas, será necessário utilizar uma estrutura de dados especial do próprio Python para garantir a eficiência das operações de inserção e remoção, conforme ilustrado abaixo.

```
1 from collections import deque
2 Q = deque() #define uma fila (deque)
3 Q.append(3) #enfileira o valor 3
4 Q.append(5) #enfileira o valor 5
5 Q.popleft() #desenfileira o próximo (valor 3)
```

Para concluir, é importante observar que ao invés de utilizar as transformações mais recomendadas para a linguagem Python de cada algoritmo (que usasse recursos específicos da linguagem, para ser mais próximo ao “jargão” da linguagem, não raro com alguns ganhos de desempenho), optou-se em escolher, dentre todas versões equivalentes de implementações em termos de correção e complexidade assintótica, aquela que fosse a mais semelhante com o algoritmo original. Isto se deve ao objetivo principal de prover um exemplo de como um algoritmo se transforma em implementação real, evitando sacrificar a fácil observação do mapeamento entre algoritmo e implementação.

## 1.7 Implementações

Nesta seção, vamos apresentar a implementação em Python, correspondente ao Algoritmo 1.3.

### 1.7.1 Algoritmo 1.3: Ordenação de Sequências

O Programa 1.2 corresponde à implementação do Algoritmo 1.3. Note que, na implementação, as condições lógicas da repetição da Linha 6 são testadas em ordem inversa em relação à ordem apresentada no pseudo-código, pois quando  $k=0$ , o valor  $s[k]$  é indefinido. Portanto, a condição  $k \geq 1$  deve ser avaliada antes. Ainda, é conveniente destacar o uso da atribuição múltipla na Linha 7 para trocar os valores de  $s[k]$  e  $s[k+1]$ .

Programa 1.2: Ordenação de sequências

```

1 def OrdenacaoSequencia(s):
2     #Dados: sequência s[1],...,s[n]
3     n = len(S)-1
4     for j in range(1,n):
5         k = j
6         while k >= 1 and s[k] > s[k+1]:
7             s[k], s[k+1] = s[k+1], s[k]
8             k = k-1

```

## 1.8 Exercícios

- 1.1 Mostrar que o conjunto  $L$  das linhas de um algoritmo e a relação  $R$  formulada por “ $sRt \Leftrightarrow$  o bloco definido por  $s$  contém o por  $t$ ” para  $s, t \in L$ , definem uma ordenação parcial. Em que condição a ordenação é total?
- 1.2 Usando a linguagem de apresentação de algoritmos, da Seção 1.3, como proceder em caso de linha continuação, de modo a não contrariar as regras relativas à estrutura da linguagem?
- 1.3 Suponha um algoritmo  $\alpha$  com  $m$  linhas e  $b$  blocos disjuntos cuja união contenha todas as linhas. Supondo  $m$  fixo, determinar a estrutura de  $\alpha$  para que (i)  $b$  seja mínimo, (ii) máximo.
- 1.4 A complexidade relativa a espaço de um algoritmo é assintoticamente não maior do que a relativa a tempo, para o mesmo algoritmo. Certo ou errado?
- 1.5 Determinar a complexidade de melhor caso do Algoritmo 1.3.
- 1.6 Escrever algoritmos para incluir e excluir elementos em uma lista encadeada, composta de  $n$  elementos. Repetir o problema supondo a lista duplamente encadeada. Qual a complexidade dos algoritmos?

- 1.7 Diz-se que uma pilha  $S$  *realiza* a permutação  $p$  de  $1, \dots, n$  quando existe uma sequência de inclusões e exclusões em  $S$ , sobre a entrada  $1, \dots, n$  tal que a ordem das exclusões corresponda a  $p$ . Caracterizar as permutações realizáveis de  $1, \dots, n$ .
- 1.8 Resolver o problema anterior para os casos em que a pilha é substituída, respectivamente, pelas seguintes listas: (i) fila, (ii) fila dupla de entrada restrita, (iii) fila dupla de saída restrita, (iv) fila dupla.
- 1.9 Uma  $k$ -pilha é uma lista em que todo elemento excluído é algum dentre os  $k$  elementos incluídos mais recentemente. Resolver o problema 1.7, com uma  $k$ -pilha substituindo a pilha. Supor, inicialmente,  $k = 2$ .

1.10 ENADE (2005)

Considere o algoritmo que implementa o seguinte processo: uma coleção desordenada de elementos é dividida em duas metades e cada metade é utilizada como argumento para a replicação recursiva do procedimento. Os resultados das duas reaplicações são, então, combinados pela intercalação dos elementos de ambas, resultando em uma coleção ordenada. Qual é a complexidade desse algoritmo?

- a)  $O(n^2)$
- b)  $O(n^{2n})$
- c)  $O(2^n)$
- d)  $O(\log n \times \log n)$
- e)  $O(n \times \log n)$

1.11 ENADE (2005)

No processo de pesquisa binária em um vetor ordenado, os números máximos de comparações necessárias para se determinar se um elemento faz parte de vetores com tamanhos 50, 1.000 e 300 são, respectivamente, iguais a

- a) 5, 100 e 30.
- b) 6, 10 e 9.
- c) 8, 31 e 18.
- d) 10, 100 e 30.
- e) 25, 500 e 150.

1.12 ENADE (2005)

No desenvolvimento de um software que analisa bases de DNA, representadas pelas letras A, C, G, T, utilizou-se as estruturas de dados: pilha e fila. Considere que, se uma sequência representa uma pilha, o topo é o elemento mais à esquerda; e se uma sequência representa uma fila, a sua frente é o elemento mais à esquerda. Analise o seguinte cenário: “a sequência inicial ficou armazenada na primeira estrutura de dados na seguinte ordem: (A,G,T,C,A,G,T,T). Cada elemento foi retirado da primeira estrutura de dados e inserido na segunda estrutura de dados, e a sequência ficou armazenada na seguinte ordem: (T,T,G,A,C,T,G,A). Finalmente, cada elemento foi retirado da segunda estrutura de dados e inserido na terceira estrutura de

dados e a sequência ficou armazenada na seguinte ordem: (T,T,G,A,C,T,G,A)”. Qual a única sequência de estruturas de dados apresentadas a seguir pode ter sido usada no cenário descrito anteriormente?

- a) Fila - Pilha - Fila.
- b) Fila - Fila - Pilha.
- c) Fila - Pilha - Pilha.
- d) Pilha - Fila - Pilha.
- e) Pilha - Pilha - Pilha.

#### 1.13 MARATONA SBC - 1<sup>a</sup> fase (2008)

Considere um número inteiro  $P$  com  $n$  dígitos decimais. Forma-se o número  $Q$  apagando-se  $d$  dígitos ( $1 \leq d < n \leq 10^5$ ) de  $P$  e mantendo os demais na mesma ordem inicial. Escreva um algoritmo para, dados  $n$ ,  $d$  e  $P$ , obter o maior  $Q$  possível.

#### 1.14 POSCOMP (2012)

As Estruturas de Dados (ED) são representadas classicamente por Tipos Abstratos de Dados (TAD), que permitem definir e especificar estas estruturas. Cada TAD pode ter diferentes tipos de operações, mas há três operações que são básicas e devem existir em qualquer TAD (além da definição de tipo de dado). Assinale a alternativa que apresenta, corretamente, essas três operações básicas.

- a) TAD de Pilha: Definição do dado (tipo utilizado) e as operações de inclusão inserção (empilhamento), remoção (desempilhamento) e impressão (apresentação dos dados).
- b) TAD de Pilha: Definição do dado (tipo utilizado) e as operações de inserção, remoção e impressão (apresentação dos dados).
- c) TAD de Fila: Definição do dado (tipo utilizado) e as operações de inserção, remoção e inicialização (criação) da estrutura.
- d) TAD de Fila: Definição do dado (tipo utilizado) e as operações de inicialização (criação), inserção e impressão (apresentação dos dados).
- e) TAD de Lista: Definição do dado (tipo utilizado) e as operações de inicialização (criação), inserção numa posição da Lista e remoção de todos os elementos da Lista (destruição da lista).

## 1.9 Notas Bibliográficas

O histórico artigo da ponte de Königsberg é de Euler (1736). As árvores foram introduzidas por Kirchhoff (1847) e Cayley (1857), respectivamente, para aplicações em circuitos elétricos e química orgânica. Contudo, foram também independentemente apresentadas em Jordan (1869), num contexto puramente matemático. Conforme mencionado, supõe-se que a formulação do problema das quatro cores seja de F. Guthrie. De Morgan escreveu a primeira contribuição técnica ao assunto. Ao longo dos anos, diversas provas incompletas ou incorretas foram produzidas. A mais famosa é a de Kempe (1879), cuja incorreção foi apontada por Heawood (1890). Não obstante, a prova de Kempe continha os elementos básicos utilizados, um século mais tarde, por Appel e Haken (1977) e (1977a) na solução do problema. O método dessa solução incluiu uso intenso de computadores. Desenvolvimentos importantes na teoria de grafos, produzidos nas primeiras décadas do século XIX foram de Kuratowski (1930), König (1936), Menger (1927), entre outros. Uma história da teoria dos grafos é Biggs, Lloyd e Wilson (1976). Turing (1936) é um trabalho pioneiro e fundamental em computabilidade. Hopcroft e Ullman (1969) é um texto conhecido em linguagens formais e autômatas. Um livro recomendado sobre teoria de computação é o de Lewis

e Papadimitriou (2009). Machtey e Young (1978) tratam da teoria geral de algoritmos. Hopcroft e Ullman (1979) abrange ambos os temas. Veja também Wood (1980). Os primeiros livros da literatura brasileira, nessa área, incluem Carvalho (1981), Lucchesi, Simon, Simon, Simon e Kowaltowski (1979), Imre Simon (1981), Istvan Simon (1979), Veloso (1979) e mais recentemente, Diverio e Menezes (2011), e Figueiredo e Lamb (2016). Um texto pioneiro de técnicas de algoritmos e suas complexidades é Aho, Hopcroft e Ullman (1974). Outros textos da época são, Baase (1978), Goodman e Hedetniemi (1977) e Horowitz e Sahni (1978). Este estudo constitui também o tema central dos livros de Hopcroft (1974), Rabin (1976), Tarjan (1978) e Weide (1977). Greene e Knuth (1982) abordam a matemática para a análise de algoritmos. Terada (1982) é um texto que comprehende estes tópicos, em língua portuguesa. Pacitti e Atkinson (1975) descrevem técnicas computacionais, também em língua portuguesa. O livro clássico em estrutura de dados é Knuth (1997). Mencionam-se ainda Berztiss (1971), Harrison (1973). Horowitz e Sahni (1976), Page e Wilson (1973), Pfaltz (1977), Standish (1980), entre outros. Os primeiros textos brasileiros em estrutura de dados e algoritmos em português são Veloso, Santos, Azeredo e Furtado (1982) e Lucena (1972). Textos mais recentes incluem Ziviani (1999), Szwarcfiter e Markenzon (2010), Ascencio e Araújo (2011), e Maculan e Campello (1994). O texto de algoritmos e grafos que originou o presente livro é Szwarcfiter (1983). O livro de Goldbach e Goldbach (2012), descreve vários algoritmos, neste mesmo tema. Textos mais específicos em língua portuguesa comprehendem Fernandes, Miyazawa, Cerioli, Feofiloff (2001), algoritmos de aproximação; Figueiredo, Lemos, Fonseca, Sá (2007), algoritmos randomizados; Santos e Souza (2015), algoritmos parametrizados. Um livro abrangente sobre representação e algoritmos em grafos é o de Spinrad (2003). Um trabalho específico, em língua portuguesa, sobre representação de grafos é Pombo (1979). Algoritmos em combinatória são tratados por Even (1973), Hu (1982), Lawler (1976), Nijenhuis e Wilf (1975), Page e Wilson (1979). Papadimitriou e Steiglitz (1982), Reingold, Nievergelt e Deo (1977) e Wells (1971). O Exercício 1.7 é baseado em Knuth (1997).

# UMA INICIAÇÃO À TEORIA DOS GRAFOS

---

## 2.1 Introdução

Serão descritos neste capítulo alguns conceitos básicos da Teoria dos Grafos. A apresentação cobre, com alguma folga, o necessário à compreensão dos problemas e algoritmos discutidos nos capítulos seguintes. A terminologia e notação utilizadas, posteriormente nos problemas, são também aqui introduzidas.

## 2.2 Os Primeiros Conceitos

Um *grafo*  $G(V, E)$  é um conjunto finito não vazio  $V$  e um conjunto  $E$  de pares não ordenados de elementos distintos de  $V$ . Dizemos que  $G$  é *trivial* quando  $|V| = 1$ . Quando necessário, se utiliza o termo *grafo não direcionado*, para designar um grafo. Os elementos de  $V$  são os *vértices* e os de  $E$  são as *arestas* de  $G$ , respectivamente. Cada aresta  $e \in E$  será denotada pelo par de vértices  $e = (v, w)$  que a forma. Nesse caso, os vértices  $v, w$  são os *extremos* (ou *extremidades*) da aresta  $e$ , sendo denominados *adjacentes*. A aresta  $e$  é dita *incidente* a ambos  $v, w$ . Duas arestas que possuem um extremo comum são chamadas de *adjacentes*. Utilizaremos a notação  $n = |V|$  e  $m = |E|$ .

Um grafo pode ser visualizado através de uma *representação geométrica*, na qual seus vértices correspondem a pontos distintos do plano em posições arbitrárias, enquanto que a cada aresta  $(v, w)$  é associada uma linha arbitrária unindo os pontos correspondentes a  $v, w$  (Figura 2.1). Para maior facilidade de exposição, é usual confundir-se um grafo com a sua representação geométrica. Isto é, no decorrer do texto será utilizado o termo *grafo*, significando também a sua representação geométrica.

A partir desta definição é possível formular o seguinte problema. Dadas duas representações geométricas, correspondem elas a um mesmo grafo? Em outras palavras, é possível fazer coincidir, respectivamente, os pontos de duas representações geométricas, de modo a preservar adjacência (ou seja, de modo a fazer também coincidir as arestas)? Formalmente, o problema pode ser enunciado da seguinte maneira. Dados dois grafos  $G_1(V_1, E_1)$ ,  $G_2(V_2, E_2)$ , com  $|V_1| = |V_2| = n$ , existe uma função unívoca  $f : V_1 \rightarrow V_2$ , tal que  $(v, w) \in E_1$  se e somente se  $(f(v), f(w)) \in E_2$ , para todo  $v, w \in V_1$ ? Em caso positivo,  $G_1$  e  $G_2$  são ditos *isomorfos entre si*. Por exemplo, as representações geométricas dos grafos  $G_1$  e  $G_2$  da Figura 2.2 podem se tornar coincidentes, mediante a aplicação da função  $f$  indicada na figura. Logo  $G_1$  e  $G_2$  são isomorfos entre si. Contudo, não existe função  $f$  que faça coincidir as representações  $G_1$  e  $G_3$ . Logo,  $G_3$  é não isomorfo a ambos  $G_1, G_2$ . Esse problema, denominado *isomorfismo de grafos*, pode naturalmente ser resolvido pela força bruta, examinando-se cada uma das  $n!$  permutações de  $V_1$  (ou seja, cada função  $f$  possível). Esse algoritmo necessita de pelo menos  $\Omega(n!)$  passos,

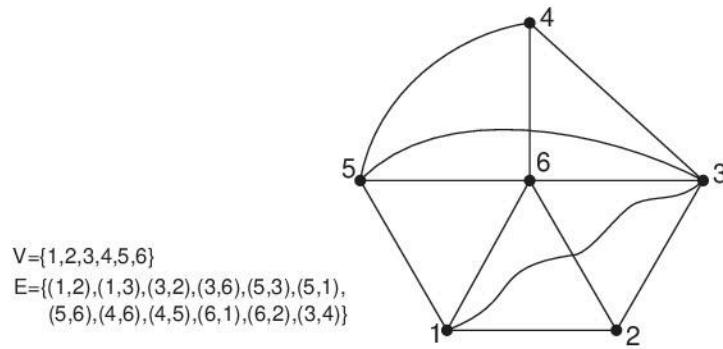
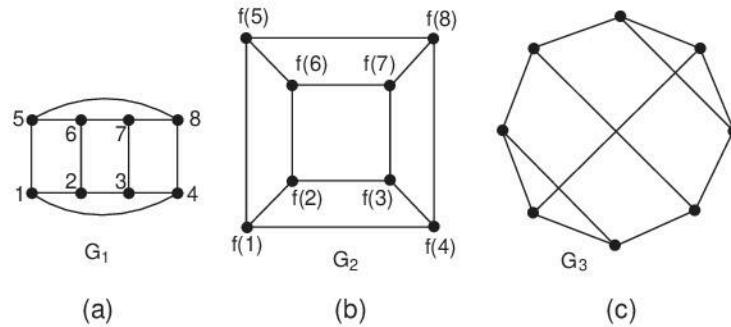
Figura 2.1: Um grafo  $(V, E)$  e uma representação geométrica do mesmo

Figura 2.2: Grafos isomorfos e não isomorfos

no pior caso. É desconhecido se existe ou não algum algoritmo eficiente para o problema geral de isomorfismo de grafos.

Algumas vezes é útil relaxar a definição de grafos, de modo a permitir uma aresta do tipo  $e = (v, v)$ , isto é, formada por um par de vértices idênticos. Uma aresta desta natureza é chamada *laço* (Figura 2.3(a)). Uma outra extensão possível consiste em substituir, na definição de grafo, o conjunto de arestas  $E$  por um multiconjunto. O efeito desta alteração é, naturalmente, permitir a existência de mais de uma aresta entre o mesmo par de vértices, as quais são então denominadas *arestas paralelas*. A estrutura  $(V, E)$  assim definida é um *multigrafo* (Figura 2.3(b)).

Em um grafo  $G(V, E)$ , define-se *grau* de um vértice  $v \in V$ , denotado por  $\text{grau}(v)$ , como sendo o número de vértices adjacentes a  $v$ . Um grafo é *regular* de *grau*  $r$ , quando todos os seus vértices possuírem o mesmo grau  $r$ . Por exemplo, cada grafo da Figura 2.2 é regular de grau 3. Observe que cada vértice  $v$  é incidente a  $\text{grau}(v)$  arestas

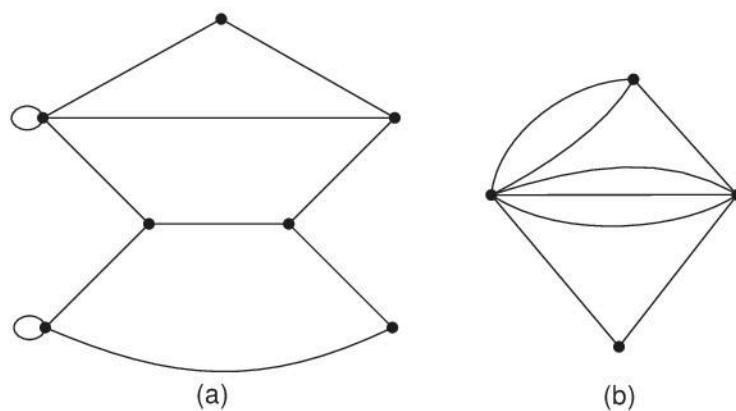


Figura 2.3: Um grafo com laços e um multigrafo

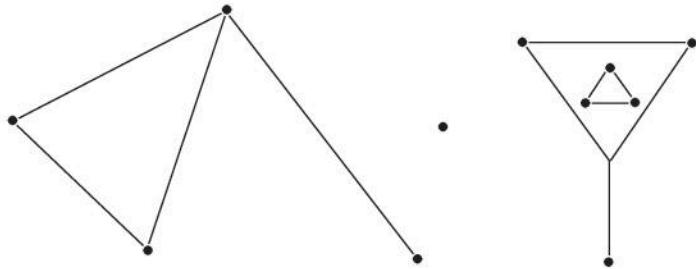


Figura 2.4: Um grafo desconexo

e cada aresta é incidente a 2 vértices. Logo,  $\sum_{v \in V} \text{grau}(v) = 2|E|$ . Um vértice que possui grau zero é chamado *isolado*.

Uma sequência de vértices  $v_1, \dots, v_k$  tal que  $(v_j, v_{j+1}) \in E$ ,  $1 \leq j < |k|$ , é denominado *caminho* ou *passeio* de  $v_1, \dots, v_k$ . Diz-se então que  $v_1$  alcança ou atinge  $v_k$ . Um caminho de  $k$  vértices é formado por  $k - 1$  arestas  $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ . O valor  $k - 1$  é o *comprimento* do caminho. Se todos os vértices do caminho  $v_1, \dots, v_k$  forem distintos, a sequência recebe o nome de *caminho simples* ou *elementar*. Se as arestas forem distintas, a sequência denomina-se *trajeto*. Por exemplo, no grafo da Figura 2.2(a), a sequência 1, 2, 3, 7 é um caminho simples, enquanto que 2, 3, 7, 6, 2, 1, 5 é um trajeto. Um *ciclo* é um caminho  $v_1, \dots, v_k, v_{k+1}$  sendo  $v_1 = v_{k+1}$  e  $k \geq 3$ . Se o caminho  $v_1, \dots, v_k$  for simples, o ciclo  $v_1, \dots, v_k, v_{k+1}$  também é denominado *simples* ou *elementar*. Um grafo que não possui ciclos simples é *acíclico*. Um *triângulo* é um ciclo de comprimento 3. Dois ciclos são considerados idênticos se um deles puder ser obtido do outro, através de uma permutação circular de seus vértices. Um caminho que contenha cada vértice do grafo exatamente uma vez é chamado *hamiltoniano*. Por outro lado, algum caminho ou ciclo que contenha cada aresta do grafo, também exatamente uma vez cada, é denominado *euleriano*. Um ciclo  $v_1, \dots, v_k, v_{k+1}$  é *hamiltoniano* quando o caminho  $v_1, \dots, v_k$  o for. Se  $G$  é um grafo que possui ciclo hamiltoniano ou euleriano, então  $G$  é denominado *hamiltoniano* ou *euleriano*, respectivamente. Por exemplo, no grafo da Figura 2.2(a), os ciclos 2, 3, 7, 6, 2 e 7, 6, 2, 3, 7 são idênticos, e o caminho 1, 5, 6, 2, 3, 7, 8, 4 é hamiltoniano. Para maior simplicidade de apresentação, quando não houver ambiguidade os termos “caminho” e “ciclo” serão utilizados com o significado de “caminho simples” e “ciclo simples”, respectivamente.

Um grafo  $G(V, E)$  é denominado *conexo* quando existe caminho entre cada par de vértices de  $G$ . Caso contrário  $G$  é *desconexo*. Observe que a representação geométrica de um grafo desconexo é necessariamente descontígua. Em especial, o grafo  $G$  será *totalmente desconexo* quando não possuir arestas.

Seja  $S$  um conjunto e  $S' \subseteq S$ . Diz-se que  $S'$  é *maximal* em relação a uma certa propriedade  $P$ , quando  $S'$  satisfaz à propriedade  $P$  e não existe subconjunto  $S'' \subset S'$ , que também satisfaz  $P$ . Observe que a definição anterior não implica necessariamente que  $S'$  seja o *maior* subconjunto de  $S$  satisfazendo  $P$ . Implica apenas o fato de que  $S'$  não está propriamente contido em nenhum subconjunto de  $S$  que satisfaça  $P$ . De maneira análoga, define-se também conjunto *minimal* em relação a uma certa propriedade. A noção de conjunto maximal (ou minimal) é frequentemente encontrada em combinatória. Ela pode ser aplicada, por exemplo, na seguinte definição. Denominam-se *componentes conexas* de um grafo  $G$  aos subgrafos maximais de  $G$  que sejam conexos. Observe que a propriedade  $P$ , nesse caso, é equivalente a “ser conexo”. As componentes conexas de um grafo  $G$  são pois os subgrafos de  $G$  correspondentes às porções contíguas de sua representação geométrica. O grafo da Figura 2.2(a) é conexo, enquanto que o da Figura 2.4 não o é. Este último possui 4 componentes conexas.

Denomina-se *distância*  $d(v, w)$  entre dois vértices  $v, w$  de um grafo ao comprimento do menor caminho entre  $v$  e  $w$ . Assim, a distância entre os vértices 1 e 8, no grafo da Figura 2.2(a) é igual a 2, isto é,  $d(1, 8) = 2$ .

Seja  $G(V, E)$  um grafo,  $e \in E$  uma aresta. Denota-se por  $G - e$  o grafo obtido de  $G$ , pela exclusão da aresta  $e$ . Se  $v, w$  é um par de vértices não adjacentes em  $G$ , a notação  $G + (v, w)$  representa o grafo obtido adicionando-se a  $G$  a aresta  $(v, w)$ . Analogamente, seja  $v \in V$  um vértice de  $G$ . O grafo  $G - v$  denota aquele obtido de  $G$  pela remoção do vértice  $v$ . Observe que excluir um vértice implica remover de  $G$  o vértice em questão e as arestas a ele incidentes. Da mesma forma,  $G + w$  representa o grafo obtido adicionando-se a  $G$  o vértice  $w$ .

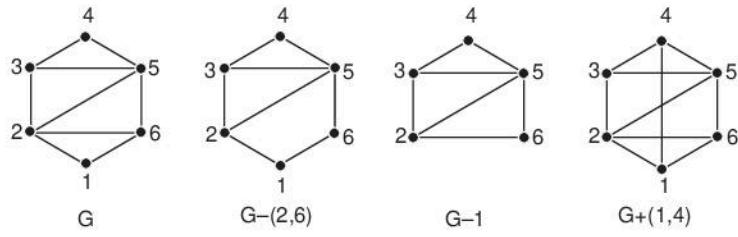


Figura 2.5: Operações de exclusão e inclusão de arestas ou vértices

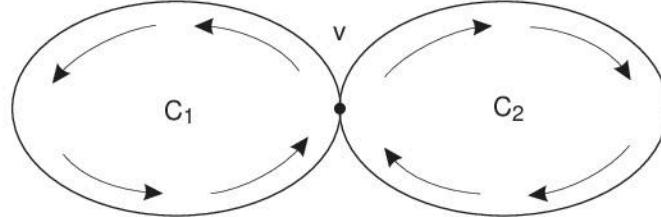


Figura 2.6: O passo principal do Teorema 2.1

Observa-se ainda que essas operações de inclusão e exclusão de vértices e arestas podem ser generalizadas. De um modo geral, se  $G$  é um grafo e  $S$  um conjunto de arestas ou vértices,  $G - S$  e  $G + S$  denotam, respectivamente, o grafo obtido de  $G$  pela exclusão e inclusão de  $S$ , respectivamente. Ver Figura 2.5.

Dado um grafo  $G$ , pode-se indagar quais seriam as condições para que  $G$  possua um ciclo euleriano. A resposta a esta questão constituiu o primeiro teorema conhecido em Teoria de Grafos, mencionado no Capítulo 1 e reproduzido a seguir.

### Teorema 2.1

*Um grafo  $G$  conexo possui ciclo euleriano se e somente se todo vértice de  $G$  possuir grau par.*

**Prova** Seja  $C$  um ciclo euleriano de  $G$ . Cada ocorrência de um dado vértice  $v$  em  $C$  contribui com 2 unidades para o cálculo do grau de  $v$ . Como cada aresta de  $G$  aparece exatamente uma vez em  $C$ , conclui-se que  $v$  possui grau par. Para a prova de suficiência, particiona-se inicialmente as arestas de  $G$  em um conjunto  $C$  de ciclos simples, do seguinte modo. Como cada vértice de  $G$  possui grau  $\geq 2$ ,  $G$  contém necessariamente algum ciclo simples  $C_1$ . Se  $C_1$  contém todas as arestas de  $G$ , o particionamento  $C$  está terminado. Caso contrário, remove-se de  $G$  as arestas do ciclo  $C_1$  e os vértices isolados, porventura formados após essa operação. No novo grafo assim obtido, cada vértice possui ainda grau par. Determina-se assim um novo ciclo simples e assim por diante. Ao final, as arestas de  $G$  encontram-se particionadas em ciclos simples. Para determinar o ciclo euleriano, considera-se um ciclo, por exemplo  $C_1$  do particionamento  $C$ . Se não há mais ciclos de  $C$  a serem considerados, a prova está concluída. Caso contrário, como  $G$  é conexo, existe um ciclo  $C_2 \in P$  tal que  $C_1$  e  $C_2$  possuem um vértice comum  $v$ . Então o ciclo formado pela união das arestas de  $C_1$  e  $C_2$  contém cada uma dessas arestas exatamente uma vez (Figura 2.6). Repete-se o processo, considerando um novo ciclo  $C_3 \in C$ , ainda não considerado e assim por diante. ■

Observe que a prova de suficiência do Teorema 2.1 é construtiva. De fato, ela corresponde a um algoritmo de complexidade  $O(n + m)$  para encontrar (se existir) um ciclo euleriano num grafo com  $n$  vértices e  $m$  arestas. Em contraste, a determinação de um ciclo hamiltoniano apresenta dificuldade muito maior. Na realidade, ainda hoje é desconhecido se existe ou não algum algoritmo eficiente para resolver este problema.

Denomina-se *complemento* de um grafo  $G(V, E)$  ao grafo  $\bar{G}$ , o qual possui o mesmo conjunto de vértices do que  $G$  e tal que para todo par de vértices distintos  $v, w \in V$ , tem-se que  $(v, w)$  é aresta de  $\bar{G}$  se e somente se não o for de  $G$ . Ver Figura 2.7.

Um grafo é *completo* quando existe uma aresta entre cada par de seus vértices. Utiliza-se a notação  $K_n$ , para designar um grafo completo com  $n$  vértices (Figura 2.8). O grafo  $K_n$  possui pois o número máximo possível de arestas para um dado  $n$ , ou seja  $\binom{n}{2}$  arestas. Um grafo  $G(V, E)$  é *bipartido* quando o seu conjunto de vértices

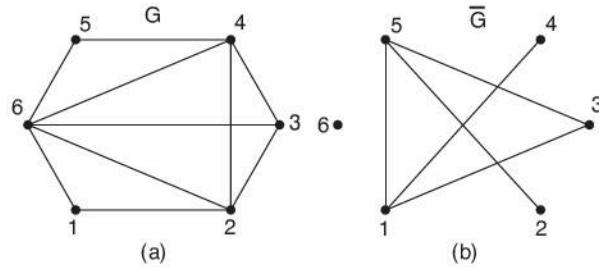


Figura 2.7: Um grafo e seu complemento

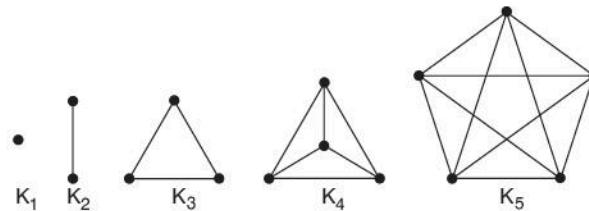


Figura 2.8: Grafos completos

$V$  puder ser partitionado em dois subconjuntos  $V_1, V_2$ , tais que toda aresta de  $G$  une um vértice de  $V_1$  a outro de  $V_2$ . É útil denotar-se o grafo bipartido  $G$  por  $(V_1 \cup V_2, E)$ . Um grafo *bipartido completo* possui uma aresta para cada par de vértices  $v_1, v_2$ , onde  $v_1 \in V_1$  e  $v_2 \in V_2$ . Sendo  $n_1 = |V_1|$  e  $n_2 = |V_2|$ , um grafo bipartido completo é denotado por  $K_{n_1, n_2}$  e obviamente possui  $n_1 n_2$  arestas. Ver Figuras 2.8 e 2.9. Pode-se indagar se existe algum processo eficiente para se reconhecer grafos bipartidos. A resposta é afirmativa, e se baseia no seguinte Teorema.

### Teorema 2.2

Um grafo  $G(V, E)$  é bipartido se e somente se todo ciclo de  $G$  possuir comprimento par.

**Prova** Seja  $v_1, \dots, v_k, v_1$  um ciclo de comprimento  $k$  do grafo bipartido  $G$  e seja  $v_1 \in V_1$ . Logo,  $v_2 \in V_2, v_3 \in V_1, v_4 \in V_2$ , e assim por diante. Como  $(v_k, v_1) \in E$  implica  $v_k \in V_2$ . Portanto  $k$  é par, o que mostra a necessidade. A prova da suficiência consiste em exibir os subconjuntos  $V_1, V_2$  que biparticionam o conjunto de vértices do grafo  $G$ , no qual todos os ciclos possuem comprimento par. Seleccione arbitrariamente um vértice  $v_1 \in V$ . Defina  $V_1$  como contendo  $v_1$  e todos os vértices que se encontram à distância par de  $v_1$ . Defina  $V_2 = V - V_1$ . Suponha que exista uma aresta  $(a, b)$  entre dois vértices  $a, b \in V_1$ . Então os caminhos mais curtos entre  $v_1$  e  $a$ , e entre  $v_1$  e  $b$  unidos com a aresta  $(a, b)$  formam um ciclo de comprimento ímpar, uma contradição. As demais possibilidades são tratadas de forma análoga. ■

Um *subgrafo*  $G_2(V_2, E_2)$  de um grafo  $G_1(V_1, E_1)$  é um grafo tal que  $V_2 \subseteq V_1$  e  $E_2 \subseteq E_1$ . Se além disso,  $G_2$  possuir toda aresta  $(v, w)$  de  $G_1$  tal que ambos  $v$  e  $w$  estejam em  $V_2$ , então  $G_2$  é o *subgrafo induzido pelo subconjunto de vértices*  $V_2$ . Diz-se então que  $V_2$  induz  $G_2$ . Ou seja, o subgrafo induzido  $G_2$  de  $G_1$  satisfaz:

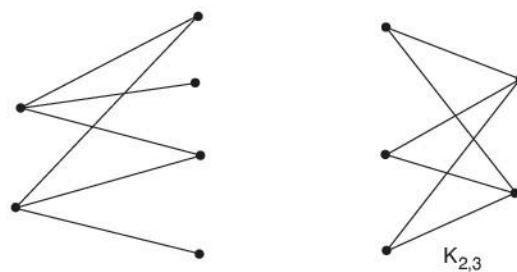


Figura 2.9: Exemplos de grafo bipartido

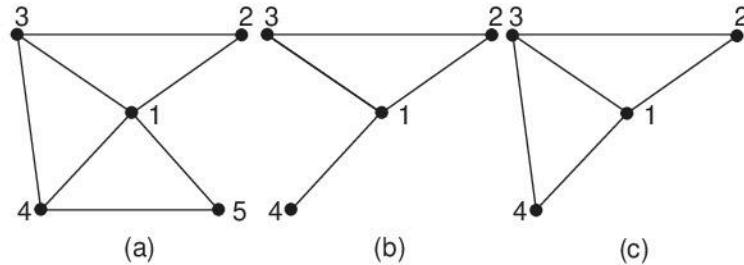


Figura 2.10: Exemplos de subgrafos

para  $v, w \in V_2$ , se  $(v, w) \in E_1$  então  $(v, w) \in E_2$ . Como exemplo, os grafos das Figuras 2.10(b) e (c) são ambos subgrafos daqueles da Figura 2.10(a). Mas somente o da 2.10(c) é induzido. Similarmente, para um dado  $E_2 \subseteq E_1$ , se  $V_2 \subseteq V_1$  for exatamente o subconjunto de vértices de  $V_1$  incidentes a alguma aresta de  $E_2$ , então  $G_2(V_2, E_2)$  é o subgrafo induzido pelo subconjunto de arestas  $E_2 \subseteq E_1$ .

Denomina-se *clique* de um grafo  $G$  a um subgrafo de  $G$  que seja completo. Chama-se *conjunto independente de vértices* a um subgrafo induzido de  $G$ , que seja totalmente desconexo. Ou seja, numa clique existe uma aresta entre cada par de vértices distintos. Num conjunto independente de vértices não há aresta entre qualquer par de vértices. O *tamanho* de uma clique ou conjunto independente de vértices é igual à cardinalidade de seu conjunto de vértices. Por exemplo, no grafo da Figura 2.7(a), o subgrafo induzido pelo subconjunto de vértices  $\{2, 3, 4, 6\}$  é uma clique de tamanho 4. O subgrafo induzido pelo subconjunto  $\{1, 3, 5\}$  é um conjunto independente de vértices, de tamanho 3. Dado um grafo  $G$ , o problema de encontrar em  $G$  uma clique ou conjunto independente de vértices com um dado tamanho  $k$ , pode ser facilmente resolvido por um processo de força bruta, através do exame do subgrafo induzido por  $\binom{n}{k}$  subconjuntos de  $V$  com  $k$  vértices. Este método corresponde pois a um algoritmo de complexidade  $\Omega(n^k)$ . É desconhecido se existe algum processo eficiente para resolver este problema.

Finalmente, um grafo será denominado *rotulado* em vértices (ou arestas) quando a cada vértice (ou aresta) estiver respectivamente associado um conjunto, denominado *rótulo*. Por exemplo, o grafo da Figura 2.10(a) pode ser considerado como rotulado em vértices, pois a estes estão associados os conjuntos  $\{1\}$ ,  $\{2\}$ ,  $\{3\}$ ,  $\{4\}$  e  $\{5\}$ , respectivamente.

## 2.3 Árvores

Um grafo que não possui ciclos é *acíclico*. Denomina-se *árvore* a um grafo  $T(V, E)$  que seja acíclico e conexo. Se um vértice  $v$  da árvore  $T$  possuir grau  $\leq 1$  então  $v$  é uma *folha*. Caso contrário, se  $\text{grau}(v) > 1$  então  $v$  é um *vértice interior*. Um conjunto de árvores é denominado *floresta*. Assim sendo, todo grafo acíclico é uma floresta. A Figura 2.11 ilustra uma floresta composta de 2 árvores. Os vértices  $v$  e  $w$  assinalados na árvore mais à esquerda da figura são respectivamente uma folha e um vértice interior. A Figura 2.12 ilustra todas as possíveis árvores (não isomorfas entre si) com 6 vértices.

Observe que toda árvore  $T$  com  $n$  vértices possui exatamente  $n - 1$  arestas. O seguinte argumento induutivo justifica a afirmativa. Se  $T$  possui 1 vértice então obviamente seu número de arestas é zero. Suponha que  $T$  contenha  $n - 1$  vértices e  $n - 2$  arestas,  $n > 1$ . Considere a adição de uma nova aresta  $e = (v, w)$ . Como  $T$  é conexo, pelo menos um dentre  $v, w$  pertence a  $T$ . Mas como  $T$  é acíclico, pelo menos um deles não pertence a  $T$  (caso contrário, a inclusão de uma nova aresta entre dois vértices de  $T$  produz um ciclo).

Logo, exatamente um dentre  $v, w$  pertence a  $T$ , o que significa que a árvore passa a possuir  $n$  vértices e  $n - 1$  arestas. Através de um raciocínio análogo pode-se mostrar que o número de folhas de  $T$  varia entre um mínimo de 2 e máximo de  $n - 1$ , para  $n > 2$ .

Árvores constituem uma classe extremamente importante de grafos, principalmente devido a sua aplicação nas mais diversas áreas. O teorema abaixo é uma caracterização para as árvores.

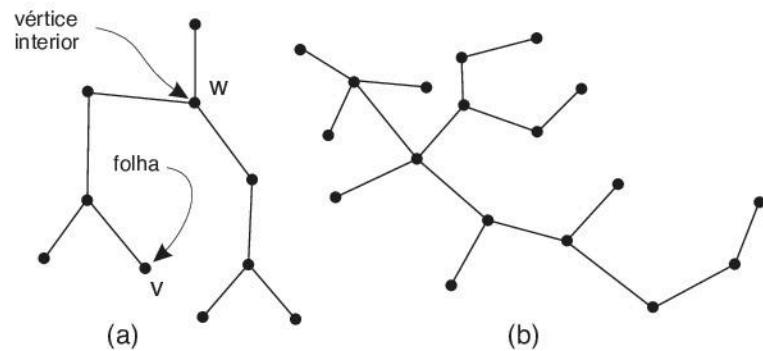


Figura 2.11: Uma floresta composta de duas árvores

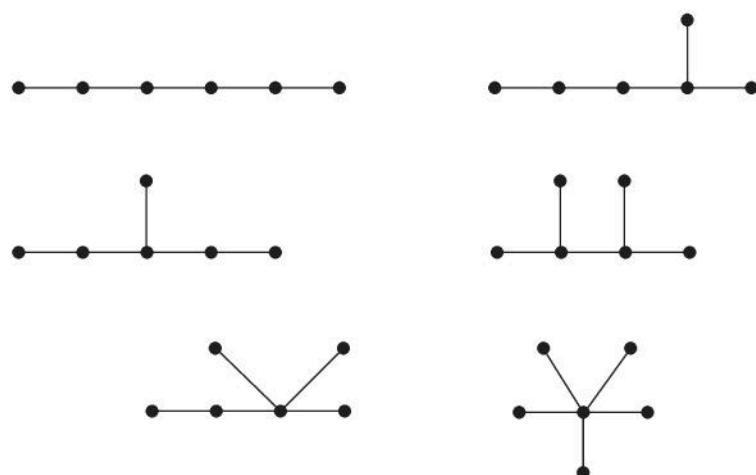


Figura 2.12: As árvores com 6 vértices

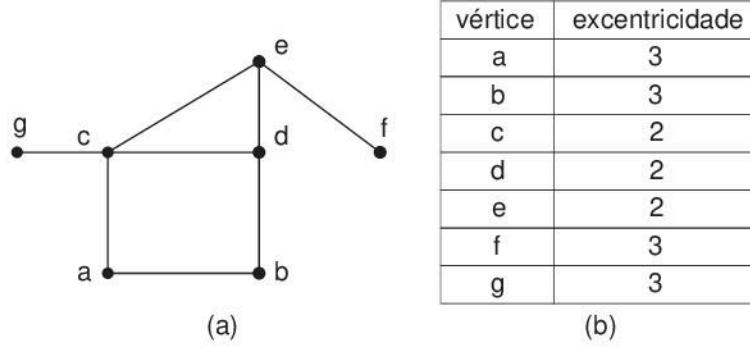


Figura 2.13: Excentricidade de vértices

**Teorema 2.3**

Um grafo  $G$  é uma árvore se e somente se existir um único caminho entre cada par de vértices de  $G$ .

**Prova** Seja  $G$  uma árvore. Então  $G$  é conexo e portanto existe pelo menos um caminho entre cada par  $v, w$  de vértices de  $G$ . Suponha que existam dois caminhos distintos  $vP_1w$  e  $vP_2w$ , entre  $v$  e  $w$ . Então  $vP_1wP_2v$  é um ciclo, o que contradiz  $G$  ser acíclico. Isto prova a necessidade. Reciprocamente, se existe exatamente um caminho entre cada par de vértices de  $G$ , então o grafo é conexo e, além disso, não pode conter ciclos. Logo  $G$  é uma árvore. ■

Além do Teorema 2.3, existem diversas outras possíveis caracterizações para as árvores. O teorema seguinte resume algumas dessas.

**Teorema 2.4**

Seja  $G(V, E)$  um grafo. As possíveis afirmativas são equivalentes.

- (i)  $G$  é uma árvore.
- (ii)  $G$  é conexo e  $|E|$  é mínimo.
- (iii)  $G$  é conexo e  $|E| = |V| - 1$ .
- (iv)  $G$  é acíclico e  $|E| = |V| - 1$ .
- (v)  $G$  é acíclico e para todo  $v, w \in V$ , a adição da aresta  $(v, w)$  produz um grafo contendo exatamente um ciclo.

A prova do Teorema 2.4 não apresenta dificuldade e será omitida. Seja  $G(V, E)$  um grafo. Denomina-se *excentricidade* de um vértice  $v \in V$  ao valor máximo da distância entre  $v$  e  $w$ , para todo  $w \in V$ . O *centro* de  $G$  é o subconjunto dos vértices de excentricidade mínima. A tabela da Figura 2.13(b) apresenta as excentricidades dos vértices do grafo da 2.13(a). O centro desse grafo é então o subconjunto dos vértices  $\{c, d, e\}$ .

O centro de um grafo pode possuir no mínimo um e no máximo  $n$  vértices. O centro de uma árvore possui não mais do que 2 vértices. O lema seguinte será utilizado na justificativa deste fato.

**Lema 2.1**

Seja  $T$  uma árvore com pelo menos 3 vértices. Seja  $T'$  a árvore obtida de  $T$  pela exclusão de todas as suas folhas. Então  $T$  e  $T'$  possuem o mesmo centro.

**Prova** Observe inicialmente que se um vértice  $f$  de  $T$  é uma folha então  $f$  não pertence ao centro de  $T'$ . Isto porque o vértice  $g$  adjacente a  $f$  possui necessariamente excentricidade uma unidade menor do que a de  $f$ . Seja agora um vértice interior  $v$  de  $T$ . O vértice  $w$ , cuja distância a  $v$  é máxima, é necessariamente uma folha. Logo a exclusão de todas as folhas de  $T$  faz decrescer de uma unidade a excentricidade de cada um de seus vértices interiores. Isto completa a prova. ■

Tem-se então:

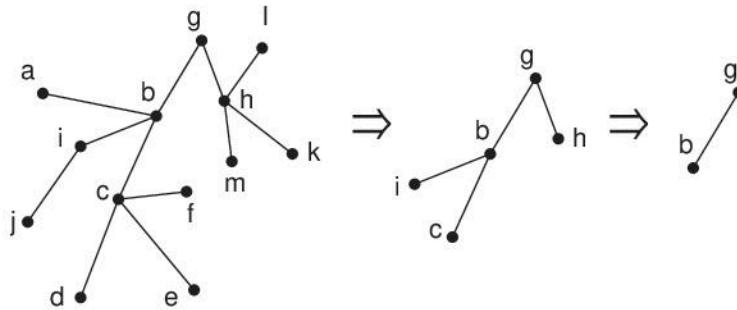


Figura 2.14: Determinação do centro de uma árvore

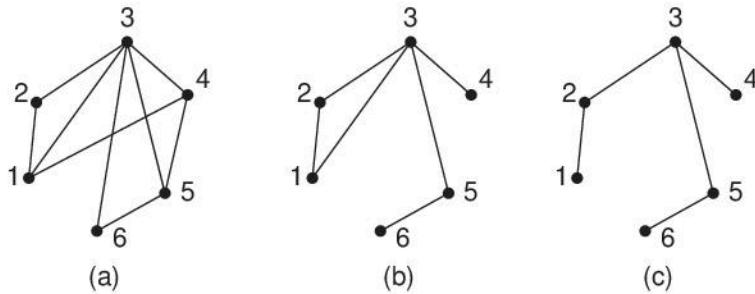


Figura 2.15: Subgrafo gerador e árvore geradora

**Teorema 2.5**

O centro de uma árvore  $T$  possui um ou dois vértices.

**Prova** Se  $T$  possui até dois vértices o teorema é trivial. Caso contrário, aplicar repetidamente o Lema 2.1. ■

Observe que a prova do Teorema 2.5 é construtiva. Ela corresponde ao seguinte algoritmo para a determinação do centro de uma árvore (ver Figura 2.14).

**Algoritmo 2.1:** Determinação do centro de uma árvore

**Dados:** árvore  $T(V, E)$

**enquanto**  $|V| > 2$  **efetuar**

excluir as folhas da árvore

Ao final do procedimento anterior,  $V$  contém o centro da árvore dada. Para avaliar a complexidade do algoritmo, observe que o teste da condição “ $|V| > 2$ ” é efetuado  $O(n)$  vezes. A identificação de todas as folhas da árvore requer  $O(n)$  operações. Logo, o algoritmo pode ser implementado em tempo  $O(n^2)$ . Contudo, observando-se a relação entre o grau de cada vértice das árvores, antes e após a operação de remoção das folhas, respectivamente, é possível uma implementação em tempo  $O(n)$ .

Denomina-se *subgrafo gerador* (ou *subgrafo de espalhamento*) de um grafo  $G_1(V_1, E_1)$  a um subgrafo  $G_2(V_2, E_2)$  de  $G_1$  tal que  $V_1 = V_2$ . Quando o subgrafo gerador é uma árvore, ele recebe o nome de *árvore geradora* (*árvore de espalhamento*). Os grafos das Figuras 2.15(b) e (c) são subgrafos geradores do grafo da 2.15(a), enquanto que o da 2.15(c) é uma árvore geradora dos outros dois.

Todo grafo conexo  $G$  possui árvore geradora. O seguinte processo construtivo de uma árvore geradora  $T$  de  $G$  justifica esta afirmativa. Considere uma aresta  $e$  de  $G$ . Remover  $e$  de  $G$  se  $G - e$  for conexo. Quando todas as arestas que permaneceram já tiverem sido consideradas, o grafo resultante é uma árvore geradora de  $G$ . Observe que se  $G$  não for conexo, pode-se aplicar esta mesma ideia a cada componente conexa de  $G$ , obtendo então uma *floresta geradora* de  $G$ , ou seja, uma árvore geradora para cada componente conexa do grafo.

Seja  $G(V, E)$  um grafo conexo e  $T(V, E)$  uma árvore geradora de  $G$ . Uma aresta  $e \in E - E_T$  é denominada *elo* de  $G$  em relação a  $T$ . Sendo  $|V| = n$ ,  $|E| = m$  tem-se:  $|E_T| = n - 1$  e portanto o número total de elos distintos

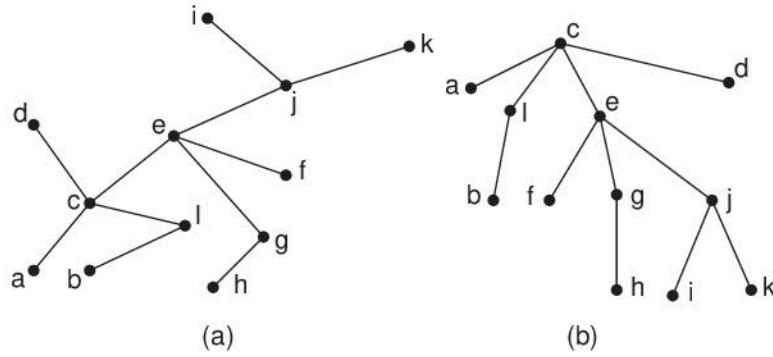


Figura 2.16: Árvores

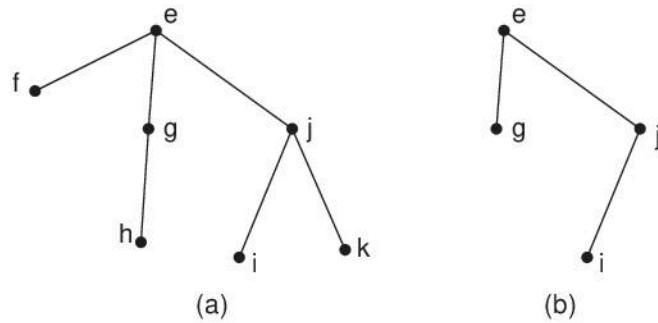


Figura 2.17: Subárvore e subárvore parcial

de  $G$  é igual a  $m - n + 1$ . O valor  $m - n + 1$  é denominado *posto* do grafo  $G$ . Observe que a adição do elo  $e$  à árvore  $T$  produz sempre um único ciclo simples, isto é, o grafo  $G + e$  possui exatamente um ciclo simples, o qual contém a aresta  $e$ . Este ciclo é denominado *ciclo fundamental* de  $G$ , em relação a  $T$ . Chama-se *conjunto fundamental de ciclos* ao conjunto de todos os ciclos fundamentais de  $G$ , em relação à árvore  $T$ . Assim sendo, um conjunto fundamental de ciclos é formado por  $m - n + 1$  ciclos simples, correspondentes respectivamente aos  $m - n + 1$  elos de  $G$  em relação a  $T$ . Cada ciclo do conjunto fundamental possui exatamente um elo. Para um certo grafo  $G$  um conjunto fundamental de ciclos depende da árvore geradora considerada. Contudo a cardinalidade desse conjunto é invariante e igual ao posto do grafo. Por exemplo, o grafo da Figura 2.15(a) possui posto igual a 4. Seus elos em relação à árvore geradora da 2.15(c) são respectivamente as arestas (1,3), (1,4), (3,6) e (4,5). O conjunto fundamental de ciclos do grafo considerado, em relação a essa árvore geradora, é constituído pelos 4 ciclos {1, 2, 3, 1; 1, 2, 3, 4, 1; 3, 5, 6, 3; 3, 4, 5, 3}.

Uma árvore  $T(V, E)$  é denominada *enraizada* quando algum vértice  $v \in V$  é escolhido como especial, sendo este vértice também chamado de *raiz* da árvore. Uma árvore não enraizada é também denominada *árvore livre*. Por exemplo, a árvore livre da Figura 2.16(a) torna-se a árvore enraizada da 2.16(b), quando o vértice  $c$  é escolhido como raiz. Sejam  $v, w$  dois vértices de uma árvore enraizada  $T$  de raiz  $r$ . Suponha que  $v$  pertença ao caminho de  $r$  a  $w$  em  $T$ . Então  $v$  é *ancestral* de  $w$ , sendo  $w$  *descendente* de  $v$ . Se ainda  $v \neq w$ ,  $v$  é *ancestral próprio* de  $w$ , e este é *descendente próprio* de  $v$ . Além disso, se  $(v, w)$  é aresta de  $T$ , então  $v$  é *pai* de  $w$ , sendo  $w$  *filho* de  $v$ . Dois vértices que possuem o mesmo pai são *irmãos*.

A raiz de uma árvore naturalmente não possui pai, enquanto que todo vértice  $v \neq r$  possui um único. Uma *folha* é um vértice que não possui filhos. Denomina-se *nível* de um vértice  $v$ , denotado por  $\text{nível}(v)$ , ao número de vértices do caminho da raiz  $r$  a  $v$ . Então  $\text{nível}(r) = 1$  e se dois vértices  $v, w$  são irmãos,  $\text{nível}(v) = \text{nível}(w)$ . A *altura* da árvore  $T$  é igual ao valor máximo de  $\text{nível}(v)$ , para todo vértice  $v$  de  $T$ . Por exemplo, o conjunto dos ancestrais do vértice  $j$  da árvore enraizada da Figura 2.16(b) é  $\{j, e, c\}$ . O conjunto dos ancestrais próprios de  $j$  é  $\{e, c\}$ . Os descendentes de  $j$  é  $\{j, i, k\}$  e  $\{i, k\}$  é o conjunto dos descendentes próprios desse vértice. O pai de  $j$  é o vértice  $e$ , enquanto que seus filhos são  $i, k$ . O nível de  $j$  é 2 e a altura da árvore é 3.

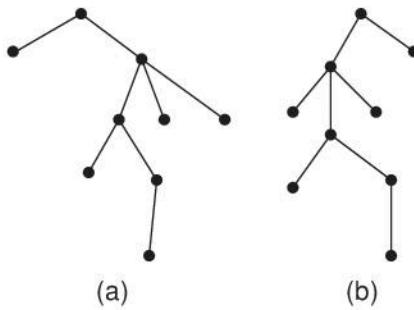
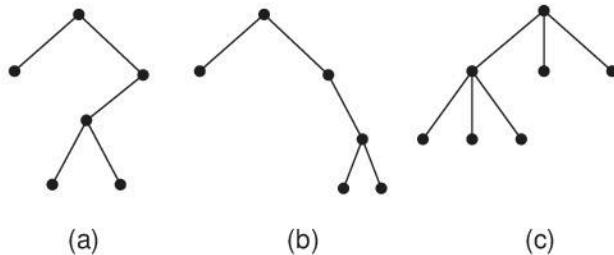


Figura 2.18: Árvores enraizadas isomorfas

Figura 2.19: Árvores  $m$ -árias

Seja  $T(V, E)$  uma árvore enraizada e  $v \in V$ . Uma *subárvore*  $T_v$  de  $T$  é a árvore enraizada cuja raiz é  $v$ , definida pelo subgrafo induzido em  $T$  pelos descendentes de  $v$ . Isto é,  $u$  é pai de  $w$  em  $T_v$  se e somente se o for em  $T$ , para todo  $u, w$  descendentes de  $v$  em  $T$ . Seja  $S$  um subconjunto de vértices de  $T_v$  tal que  $T_v - S$  seja conexo e  $v \notin S$ . O grafo  $T_v - S$  é então denominado *subárvore parcial* de raiz  $v$ . Obviamente a subárvore de raiz  $v$  é única para cada  $v \in V$ , enquanto que a subárvore parcial não o é. A árvore da Figura 2.17(a) é a subárvore de raiz  $e$  da árvore enraizada da 2.16(b), enquanto que a da 2.17(b) é uma subárvore parcial da raiz  $e$ , daquela mesma árvore.

Observe que na definição da árvore enraizada é irrelevante a ordem em que os filhos de cada vértice  $v$  são considerados. Caso esta ordenação seja relevante, a árvore é denominada *enraizada ordenada*. Assim sendo, numa árvore enraizada ordenada, para cada vértice  $v$  que possui filhos, pode-se identificar o primeiro filho de  $v$  (o mais à esquerda), o segundo (segundo mais à esquerda) e assim por diante. Obviamente, se duas árvores enraizadas são isomórficas, não necessariamente elas o serão como árvores enraizadas ordenadas. Para tanto, o isomorfismo deve preservar também a ordenação. As árvores das Figuras 2.18(a) e (b) são isomórficas entre si, se consideradas como enraizadas. Mas não o são como enraizadas ordenadas. Observe, por outro lado, que árvores isomórficas livres também não serão necessariamente isomórficas, se consideradas como enraizadas. Há diversas aplicações em árvores, que requerem a mencionada ordenação.

Uma árvore *estritamente  $m$ -ária*  $T$  é uma árvore enraizada ordenada em que cada vértice não folha possui exatamente  $m$  filhos,  $m \geq 1$ . Quando  $m = 2$  a árvore é *estritamente binária*. A cada filho  $w$  de todo vértice não folha  $v$  de  $T$ , atribua agora um rótulo inteiro positivo  $r(w)$ ,  $1 \leq r(w) \leq m$ , igual à posição que  $w$  ocupa na ordenação dos filhos de  $v$ . Uma subárvore parcial de uma árvore estritamente  $m$ -ária que possui essa rotulação é chamada *árvore  $m$ -ária*. Ou seja, numa árvore  $m$ -ária  $T$  dois vértices irmãos  $w_j, w_{j+1}$  consecutivos na ordenação dos filhos de um vértice  $v$  podem ter rótulos não consecutivos (não necessariamente  $r(w_{j+1}) - r(w_j) = 1$ ). Pode-se então considerar que uma árvore  $m$ -ária é obtida a partir de uma estritamente  $m$ -ária pela remoção de  $r(w_{j+1}) - r(w_j) - 1$  subárvores entre cada par de irmãos  $w_j, w_{j+1}$ , além de  $r(w_1) - 1$  subárvores antes do filho  $w_1$  de  $v$  e  $r(w_f) - 1$  após o último filho  $w_f$  de  $v$ .

Por exemplo, numa árvore binária cada filho  $w$  de  $v$  pode ser identificado como o *filho esquerdo* ou *direito*. Naturalmente, o filho esquerdo pode existir sem o direito, ou vice-versa. As Figuras 2.19(a) e (b) ilustram árvores binárias. Elas não são isomórficas, apesar de o serem como árvores enraizadas ordenadas. A árvore da Figura 2.19(c) é estritamente ternária.

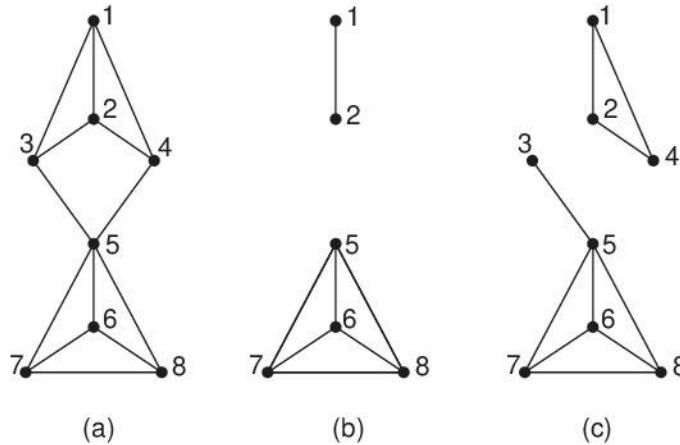


Figura 2.20: Cortes de um grafo

## 2.4 Conectividade

Seja  $G(V, E)$  um grafo conexo. Um *corte de vértices* de  $G$  é um subconjunto minimal de vértices  $V' \subseteq V$  cuja remoção de  $G$  o desconecta ou o transforma no grafo trivial. Assim sendo, o grafo  $G - V'$  é desconexo ou trivial e para todo subconjunto próprio  $V'' \subset V'$ ,  $G - V''$  é conexo e não trivial. Analogamente, um *corte de arestas* de  $G$  é um subconjunto minimal de arestas  $E' \subseteq E$ , cuja remoção de  $G$  o desconecta. Isto é,  $G - E'$  é desconexo e para todo subconjunto próprio  $E'' \subset E'$ ,  $G - E''$  é conexo. Se  $G$  é um grafo completo  $K_n$ ,  $n > 1$ , então não existe subconjunto próprio de vértices  $V' \subset V$  cuja remoção desconecte  $G$ , mas removendo-se  $n - 1$  vértices resulta o grafo trivial. Denomina-se *conectividade de vértices*  $c_V$  de  $G$  à cardinalidade do menor corte de vértices de  $G$ . Analogamente, a *conectividade de arestas*  $c_E$  de  $G$  é igual à cardinalidade do menor corte de arestas de  $G$ . Ou seja,  $c_V$  é igual ao menor número de vértices cuja remoção desconecta  $G$  ou o transforma no grafo trivial. Se  $G$  é um grafo desconexo, então  $c_V = c_E = 0$ . Considere agora o grafo da Figura 2.20(a), por exemplo. O subconjunto  $\{3, 4\}$  é um corte de vértices, pois sua remoção desconecta o grafo nas componentes ilustradas na Figura 2.20(b). Em relação a este mesmo grafo, o subconjunto de arestas  $\{(1, 3), (2, 3), (4, 5)\}$  é um corte de arestas, porque removendo-o de  $G$  produz o grafo desconexo da 2.20(c). Observe também que removendo de  $G$  o subconjunto de vértices  $\{3, 4, 7\}$  desconecta  $G$ . Contudo,  $\{3, 4, 7\}$  não é um corte de vértices, pois contém propriamente o corte  $\{3, 4\}$ . Por outro lado, os cortes de vértices e arestas, de cardinalidade mínima de  $G$ , são respectivamente os subconjuntos  $\{5\}$  e  $\{(3, 5), (4, 5)\}$ . Portanto as conectividades de vértices e arestas desse grafo ilustrado na Figura 2.20 são respectivamente iguais a 1 e 2.

Sendo  $k$  um inteiro positivo, diz-se que um grafo  $G$  é  *$k$ -conexo em vértices* quando a sua conectividade de vértices é  $\geq k$ . Analogamente,  $G$  é  *$k$ -conexo em arestas* quando sua conectividade de arestas é  $\geq k$ . Em outras palavras, se  $G$  é  $k$ -conexo em vértices (arestas) então não existe corte de vértices (arestas) de tamanho  $< k$ . O grafo da Figura 2.20(a) é 1-conexo em vértices e em arestas, e 2-conexo em arestas.

Sejam  $G(V, E)$  um grafo e  $E' \subseteq E$  um corte de arestas de  $G$ . Então é sempre possível encontrar um corte de vértices  $V'$  de tamanho  $|V'| \leq |E'|$ . Basta escolher para  $V'$  precisamente o subconjunto de vértices  $v$  tais que toda aresta  $e \in E'$  é incidente a algum  $v \in V'$ . Consequentemente, para todo grafo vale  $c_V \leq c_E$ . Seja  $w$  o vértice de grau mínimo no grafo  $G$ , suposto não completo. Então é possível desconectar  $G$ , removendo do grafo as graus( $w$ ) arestas, aquelas incidentes a  $w$ . Então  $\text{grau}(w) \geq c_E \geq c_V$ . Observe que se  $G$  for o grafo completo  $K_n$ , então  $\text{grau}(w) = c_E = c_V = n - 1$ .

O estudo de conectividade constitui um tópico básico em grafos. Os grafos 1-conexos foram denominados conexos na Seção 2.2. Grafos 2-conexos (também denominados biconexos) apresentam propriedades interessantes. Considere  $G(V, E)$  um grafo. Um vértice  $v$  é denominado *articulação* quando sua remoção de  $G$  o desconecta. Ou seja,  $G - v$  é desconexo. Uma aresta  $e \in E$  é chamada *ponte* quando sua remoção de  $G$  o desconecta. Nesse caso,  $G - e$  é desconexo. Assim sendo, um grafo é biconexo em vértices (arestas) se e somente se não possuir articulações (pontes). Denominam-se *componentes biconexas* do grafo  $G$  aos subgrafos maximais de  $G$

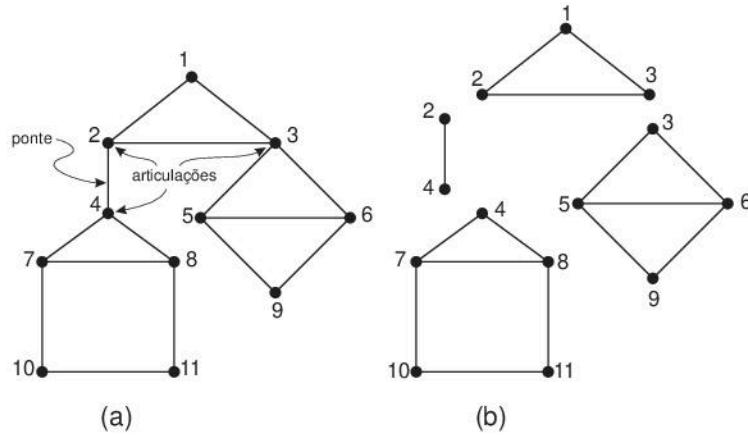


Figura 2.21: Um grafo e seus blocos

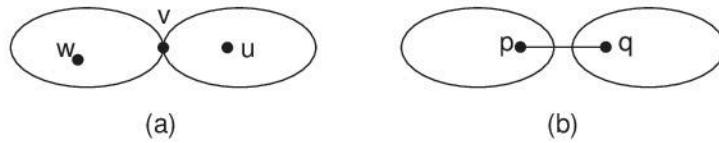


Figura 2.22: Prova do Lema 2.3

que sejam biconexos em vértices, ou isomorfos a  $K_2$ . Cada componente biconexa é também chamada *bloco* do grafo. Assim se  $G$  é biconexo em vértices então  $G$  possui um único bloco, que coincide com o próprio  $G$ . No grafo da Figura 2.21(a), os vértices 2, 3 e 4 são articulações, enquanto que a aresta (2,4) é uma ponte. Suas componentes biconexas estão indicadas na Figura 2.21(b). O lema seguinte traz informações sobre os blocos de um grafo.

### Lema 2.2

Seja  $G(V, E)$  um grafo. Então:

- (i) Cada aresta de  $G$  pertence a exatamente um bloco do grafo.
- (ii) Um vértice  $v$  de  $G$  é articulação se e somente se  $v$  pertencer a mais de um bloco do grafo.

A prova é simples e será omitida. As articulações e pontes de um grafo podem ser caracterizadas pelo Lema 2.3.

### Lema 2.3

Seja  $G(V, E)$  um grafo conexo,  $|V| > 2$ . Então:

- (i) Um vértice  $v \in V$  é articulação de  $G$  se e somente se existirem vértices  $w, u \neq v$  tais que  $v$  está contido em todo caminho entre  $w$  e  $u$  em  $G$ .
- (ii) Uma aresta  $(p, q) \in E$  é ponte se e somente se  $p, q$  for o único caminho simples entre  $p$  e  $q$  em  $G$ .

### Prova

- (i) Se  $v$  é articulação de  $G$  então  $G - v$  é desconexo. Sejam  $w, u$  dois vértices localizados em componentes conexas distintas de  $G - v$ . Então todo caminho entre  $w$  e  $u$  contém  $v$  (Figura 2.22(a)). Analogamente segue a recíproca.
- (ii) Se  $(p, q) \in E$  é uma ponte então o grafo  $G - (p, q)$  é desconexo, localizando-se  $p$  e  $q$  em componentes conexas distintas de  $G - (p, q)$ . Logo  $(p, q)$  é o único caminho simples entre  $p$  e  $q$  em  $G$  (Figura 2.22(b)). Da mesma forma, a recíproca é análoga.

**Lema 2.4**

Um grafo  $G(V, E)$ ,  $|V| > 2$ , é biconexo se e somente se cada par de vértices de  $G$  está contido em algum ciclo.

**Prova** Segue da parte (i) do Lema 2.3. ■

O Lema 2.4 pode ser generalizado, como se segue.

**Teorema 2.6**

Seja  $G$  um grafo  $k$ -conexo. Então existe algum ciclo de  $G$  passando por cada subconjunto de  $k$  vértices.

Observe que a recíproca do Teorema 2.6 não é válida para  $k > 2$ . O grafo definido por exatamente um ciclo de comprimento  $|V|$  explica a razão.

O teorema seguinte caracteriza grafos  $k$ -conexos.

**Teorema 2.7**

Um grafo  $G(V, E)$  é  $k$ -conexo se e somente se existissem  $k$  caminhos disjuntos (exceto nos extremos) entre cada par de vértices de  $G$ .

As provas dos Teoremas 2.6 e 2.7 são mais elaboradas e não serão aqui apresentadas.

## 2.5 Planaridade

Nesta seção retorna-se à noção de representação geométrica dos grafos. Seja  $G$  um grafo e  $R$  uma representação geométrica de  $G$  em um plano. A representação  $R$  é chamada *plana* quando não houver cruzamento de linhas em  $R$ , a não ser nos vértices, naturalmente. Um grafo é dito *planar* quando admitir alguma representação plana. Por exemplo, a Figura 2.23(a) ilustra uma representação não plana do grafo completo de 4 vértices  $K_4$ . A Figura 2.23(b) mostra contudo uma representação plana desse grafo. Logo, conclui-se que  $K_4$  é planar. Este conceito pode ser estendido para outras superfícies. Assim, de um modo geral diz-se que o grafo  $G$  é *imersível* em uma superfície  $S$  se existir uma representação geométrica  $R$  de  $G$ , desenhada sobre  $S$ , tal que duas linhas de  $R$  não se cruzem, a não ser nos vértices.

Seja  $G$  um grafo planar e  $R$  uma representação plana de  $G$ , num plano  $P$ . Então as linhas de  $R$  dividem  $P$  em regiões, as quais são denominadas *faces* de  $R$ . Observe que existe exatamente uma região não limitada. Esta é chamada face externa. A representação da Figura 2.23(b), por exemplo, possui 4 faces, sendo a face número 4 a externa. Segue-se que duas representações planas de um grafo possuem sempre o mesmo número de faces.

Logo este número constitui também um invariante de um grafo, sendo denotado por  $f$ . Os números de vértices, arestas e faces de um grafo planar não são independentes. O teorema seguinte os relaciona.

**Teorema 2.8**

Seja  $G$  um grafo planar. Então  $n + f = m + 2$ . Uma representação plana conveniente de  $G$  corresponde a um poliedro com  $n$  vértices,  $m$  arestas e  $f$  faces. A expressão anterior é a fórmula de Euler para poliedros e pode ser demonstrada por indução.

Observe que quanto maior é o número de arestas de um grafo  $G$  em relação ao seu número de vértices, mais difícil intuitivamente se torna a obtenção de uma representação geométrica plana para  $G$ . De fato, há um limite máximo para o número de arestas de um grafo planar, dado pelo lema a seguir.

**Lema 2.5**

Seja  $G$  um grafo planar. Então  $m \leq 3n - 6$ .

**Prova** Cada face é delimitada por um mínimo de 3 arestas e cada aresta pertence a exatamente duas faces. Logo  $2m \geq 3f$ . Substituindo na fórmula de Euler, tem-se:

$$f = m - n + 2 \Rightarrow \frac{2}{3}m \geq m - n + 2 \Rightarrow m \leq 3n - 6.$$
 ■

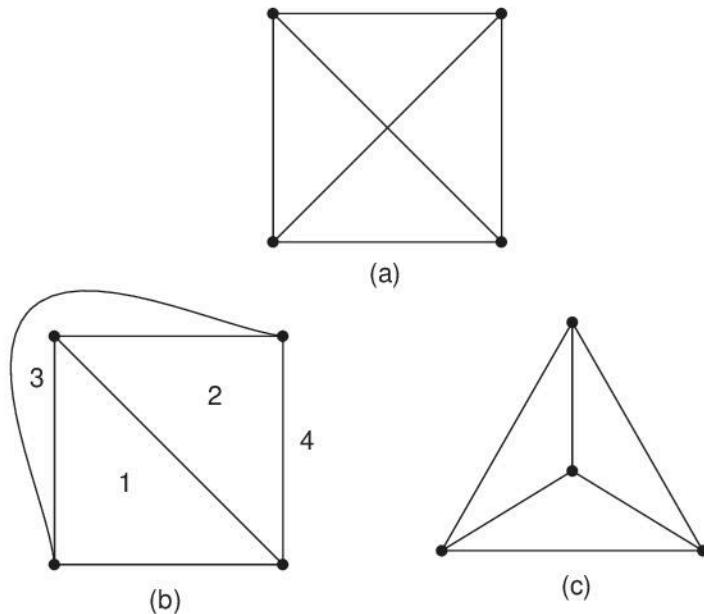


Figura 2.23: Um grafo planar

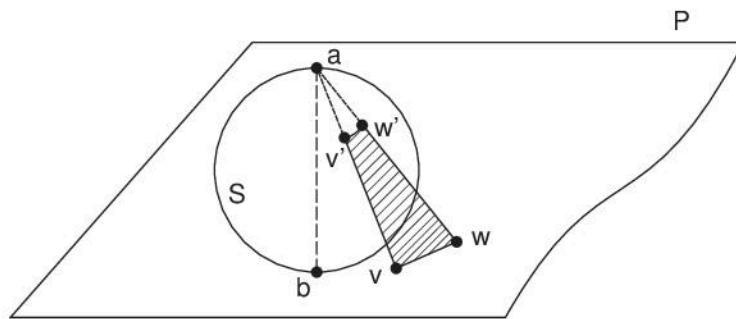


Figura 2.24: Imersão de um grafo planar na esfera

Uma outra questão relacionada é “dado um grafo  $G$  planar, admite ele uma representação plana em que todas as linhas são retas?”. Por exemplo, a representação plana da Figura 2.23(b) contém uma linha não reta. Mudando-se a disposição dos pontos obtém-se a representação da Figura 2.23(c), em que todas as linhas o são. Este fato é geral. Ou seja, todo grafo planar admite uma representação plana em que todas as linhas são retas. A prova deste fato é elaborada e não será apresentada, no texto.

Conforme foi mencionado, toda representação plana de um grafo  $G$  contém uma face externa, não limitada. Para evitar a distinção entre faces limitadas e não limitadas, pode-se considerar a representação geométrica de  $G$  em uma esfera, ao invés de num plano. Para tal, seja uma esfera  $S$  apoiada num plano  $P$ , sendo  $b$  o ponto de contato entre essas superfícies. Seja  $a$  o ponto de  $S$  diametralmente oposto a  $b$ . Denote por  $R$  uma representação plana do grafo  $G$  no plano  $P$ . Então cada ponto  $v$  de  $R$  em  $P$  possui um correspondente  $v'$  na esfera  $S$ , sendo  $v'$  definido pela interseção de  $S$  com a reta  $av$ . A Figura 2.24 mostra a representação da aresta  $(v, w)$  em uma esfera. É imediato verificar que essa construção define uma representação geométrica de  $G$  em  $S$ , sem cruzamento de linhas (a não ser nos vértices). E reciprocamente, ou seja, um grafo  $G$  é planar se e somente se  $G$  for imersível em uma esfera. As representações geométricas na esfera apresentam a vantagem de não possuírem regiões (faces) ilimitadas.

Uma questão básica é naturalmente caracterizar os grafos planares. Inicialmente tem-se:

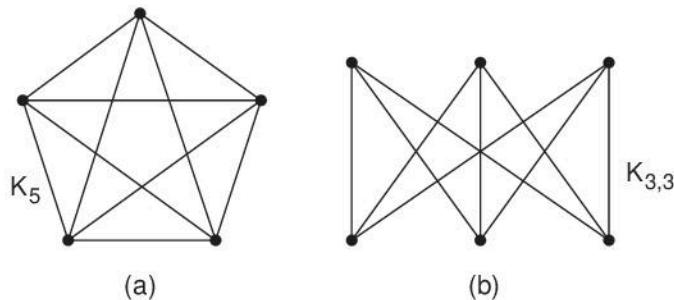
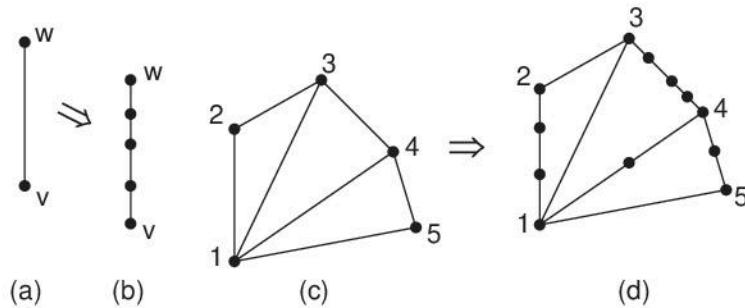
Figura 2.25: Os grafos  $K_5$  e  $K_{3,3}$ 

Figura 2.26: Subdivisão de arestas

**Lema 2.6**

Os grafos  $K_5$  e  $K_{3,3}$  são não planares.

**Prova** Para  $K_5$  tem-se  $n = 5$  e  $m = 10$ . Assim  $m > 3n - 6$ . Portanto  $K_5$  não satisfaz à condição necessária do Lema 2.5, logo não pode ser planar. Para  $K_{3,3}$  tem-se  $n = 6$  e  $m = 9$ .

Suponha que  $K_{3,3}$  seja planar. Então deve satisfazer à Fórmula de Euler, ou seja, seu número de faces é  $f = m - n + 2 = 5$ . Em qualquer representação plana de  $K_{3,3}$ , cada face deve ter pelo menos 4 arestas (pois o menor ciclo em  $K_{3,3}$  tem comprimento 4). Além disso, cada aresta pertence a exatamente 2 faces. Logo  $2m \geq 4f$ , o que contradiz  $m = 9, f = 5$ . ■

Observe que  $K_5$  e  $K_{3,3}$  (Figura 2.25) são os grafos não planares com menor número de vértices e arestas, respectivamente.

Denomina-se *subdivisão de uma aresta*  $(v, w)$  de um grafo  $G$  a uma operação que transforma  $(v, w)$  num caminho  $v, z_1, \dots, z_k, w$ , sendo  $k \geq 0$ , onde os  $z_i$  são vértices de grau 2 adicionados a  $G$ . As Figuras 2.26(a) e (b) ilustram uma subdivisão de aresta. Diz-se que um grafo  $G_2$  é uma *subdivisão* de um grafo  $G_1$ , quando  $G_2$  puder ser obtido de  $G_1$ , através de uma sequência de subdivisões de arestas de  $G_1$ . O grafo da Figura 2.26(d) é uma subdivisão daquele da 2.26(c), por exemplo.

O Teorema 2.9 é o fundamental em planaridade. Ele resolve o problema básico de caracterização dos grafos planares.

**Teorema 2.9**

Um grafo é planar se e somente se não contém como subgrafo uma subdivisão de  $K_5$  ou  $K_{3,3}$ .

A prova da necessidade é imediata. Os grafos  $K_5$  e  $K_{3,3}$  não são planares (Lema 2.6). Consequentemente qualquer subdivisão dos mesmos também não o é. Contudo a prova de suficiência é bastante envolvente e não será apresentada. O Teorema 2.9 concede aos grafos  $K_5$  e  $K_{3,3}$  um papel todo especial em planaridade.

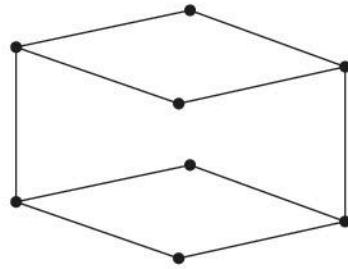


Figura 2.27: Um grafo biconexo não hamiltoniano

A partir do Teorema 2.9, construíram-se os primeiros algoritmos para reconhecer grafos planares. Atualmente são conhecidos algoritmos de complexidade  $O(n)$ , para esta finalidade.

## 2.6 Ciclos Hamiltonianos

Conforme mencionado, um grafo hamiltoniano é aquele que possui ciclo hamiltoniano, isto é, um ciclo que contém cada vértice do grafo exatamente uma vez. Por exemplo, o grafo da Figura 2.26(c) é hamiltoniano, pois contém o ciclo 1, 2, 3, 4, 5, 1. Por outro lado, as Figuras 2.27, 2.26(d) e 2.20(a) ilustram grafos não hamiltonianos.

Ao contrário dos problemas tratados nas duas seções anteriores, planaridade e conectividade, respectivamente, para o presente caso não é conhecida uma caracterização satisfatória, em termos de condições necessárias e suficientes para existência de tal ciclo. Isto se traduz no fato de que não é conhecido algoritmo eficiente para resolver o problema correspondente, de reconhecer grafos hamiltonianos. Assim sendo, os resultados ora existentes são de natureza parcial.

Todo grafo hamiltoniano é biconexo (em vértices). Isto decorre do fato de que as arestas de um ciclo hamiltoniano garantem a existência de dois caminhos disjuntos entre cada par de vértices. A recíproca não é verdadeira, conforme mostra a Figura 2.27. Uma condição necessária mais forte do que essa é a seguinte.

### Teorema 2.10

Seja  $G(V, E)$  um grafo hamiltoniano e  $S$  um subconjunto próprio de  $V$ . Então o número de componentes conexas de  $G - S$  é  $\leq |S|$ .

**Prova** Seja  $C$  um ciclo hamiltoniano de  $G$ . Então o número de componentes conexas do grafo  $C - S$  é  $\leq |S|$  (porque  $C - S$  é a união de, no máximo,  $|S|$  caminhos simples disjuntos). Então,  $C - S$  e  $G - S$  possuem o mesmo conjunto de vértices. Além disso, toda aresta de  $C - S$  é também de  $G - S$ . Logo, o número de componentes conexas desse último é  $\leq$  ao do primeiro. Isto prova o teorema. ■

Uma condição suficiente clássica é a seguinte:

### Teorema 2.11

Seja  $G(V, E)$  um grafo com pelo menos 3 vértices e tal que  $\text{grau}(v) \geq n/2$ , para todo vértice  $v \in V$ . Então  $G$  é hamiltoniano.

**Prova** Suponha o teorema falso. Então existe um grafo  $G$  maximal não hamiltoniano que satisfaz às condições da hipótese. Como  $n \geq 3$ ,  $G$  não é completo. Existem portanto  $v, w \in V$  não adjacentes. Como  $G$  é maximal,  $G + (v, w)$  é hamiltoniano. Por isso e porque  $G$  é não hamiltoniano, todo ciclo hamiltoniano de  $G + (v, w)$  contém a aresta  $(v, w)$ . Então  $G$  possui um caminho hamiltoniano  $v_1, v_2, \dots, v_n$  entre  $v_1 = v$  e  $v_n = w$ . Porque  $\text{grau}(v), \text{grau}(w) \geq n/2$ , existem necessariamente vértices  $v_j$  e  $v_{j+1}$ , para algum  $1 \leq j < n$ , tais que  $(v, v_{j+1}), (w, v_j) \in E$ . Retirando de  $C$  a aresta  $(v_j, v_{j+1})$  e acrescentando  $(v, v_{j+1})$  e  $(w, v_j)$ , transforma o caminho em ciclo hamiltoniano (Figura 2.28). Isto contradiz  $G$  ser não hamiltoniano. ■

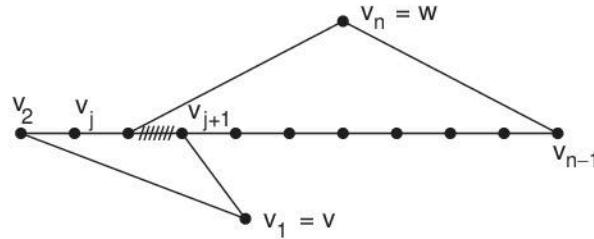


Figura 2.28: Prova do Teorema 2.10

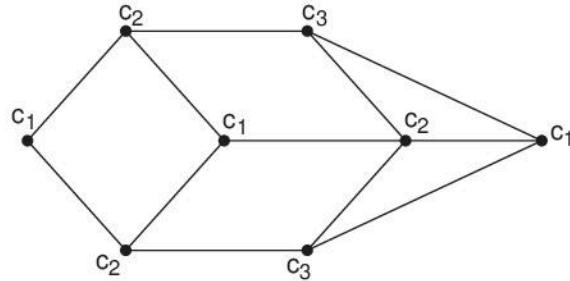


Figura 2.29: Coloração de grafos

## 2.7 Coloração

Seja  $G(V, E)$  um grafo e  $C = \{c_i\}$  um conjunto de cores. Uma *coloração* de  $G$  é uma atribuição de alguma cor de  $C$  para cada vértice de  $V$ , de tal modo que a dois vértices adjacentes sejam atribuídas cores diferentes. Assim sendo, uma coloração de  $G$  é uma função  $f : V \rightarrow C$  tal que para cada par de vértices  $v, w \in V$  tem-se  $(v, w) \in E \Rightarrow f(v) \neq f(w)$ . Uma  $k$ -coloração de  $G$  é uma coloração que utiliza um total de  $k$  cores. Isto é, uma  $k$ -coloração de  $G$  é uma coloração  $f$  cuja cardinalidade do conjunto imagem é igual a  $k$ . Diz-se então que  $G$  é  $k$ -colorível. Denomina-se *número cromático*  $\chi(G)$  de um grafo  $G$  ao menor número de cores  $k$ , para o qual existe uma  $k$ -coloração de  $G$ . Uma coloração que utiliza o número mínimo de cores é chamada *mínima*. O número cromático do grafo da Figura 2.29 é igual a 3. A figura ilustra uma 3-coloração, a qual é mínima.

Observe que é muito fácil *colorir* (isto é, obter uma coloração) um grafo. Basta utilizar um total de  $n$  cores, uma para cada vértice. Isto obviamente garante cores diferentes a vértices adjacentes. Contudo o problema de encontrar um processo eficiente para encontrar uma coloração mínima é bastante difícil. A exemplo do caso da seção anterior, não é conhecida caracterização razoável para grafos  $k$ -coloríveis. Não é conhecido, pois, algoritmo eficiente para determinar o número cromático de um grafo. De fato, há fortes indícios (Capítulo 9) de que não existe algoritmo desta natureza. Contudo, o caso especial de bicoloração, pode ser resolvido de forma simples.

### Lema 2.7

Um grafo  $G(V, E)$  é bicolorível se e somente se for bipartido.

**Prova** Se  $G$  é bipartido sejam  $V_1, V_2 \subseteq V$  os subconjuntos de  $V$  que o biparticionam. Então atribua a cor  $c_1$  aos vértices de  $V_1$  e a cor  $c_2$  aos de  $V_2$ . Reciprocamente se  $G$  é bicolorível, considere uma bicoloração de  $G$ , com cores  $c_1$  e  $c_2$ . Sejam  $V_1$  e  $V_2$  os subconjuntos de vértices que possuem as cores  $c_1$  e  $c_2$ , respectivamente. Então  $V_1, V_2$  biparticionam  $G$ . ■

O Lema 2.7 caracteriza grafos bicoloríveis e sugere um algoritmo eficiente para encontrar uma bicoloração, se existir. Se o número cromático é maior do que 2, contudo, as coisas se tornam substancialmente mais difíceis.

Os conceitos de coloração, clique e conjunto independente de vértices estão naturalmente relacionados. Por exemplo, como são necessárias  $p$  cores para colorir os  $p$  vértices de uma clique de tamanho  $k$ , conclui-se que o número cromático de um grafo  $G$  é maior ou igual do que o tamanho da maior clique de  $G$ . Considere agora uma  $k$ -coloração de  $G(V, E)$ . Sejam  $V_1, V_2, \dots, V_k$  os subconjuntos (disjuntos) de  $V$ , induzidos pela  $k$ -coloração.

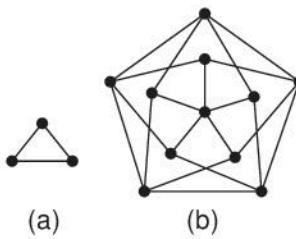
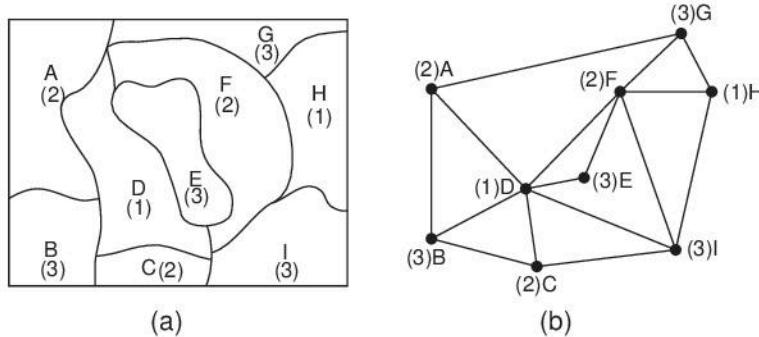
Figura 2.30: Grafos  $k$ -críticos

Figura 2.31: Coloração de mapas

Então obviamente  $\bigcup V_i = V$  e cada  $V_i$  é um conjunto independente de vértices. Então o problema de determinar uma coloração mínima de  $G$  pode ser formulado em termos de partitionar  $V$  em um número mínimo de conjuntos independentes de vértices.

Um grafo  $G$  é denominado  *$k$ -crítico* quando  $\chi(G) = k$  e para todo subgrafo próprio  $H$  de  $G$ ,  $\chi(H) < k$ . Por exemplo, os grafos das Figuras 2.30(a) e (b) são respectivamente 3-crítico e 4-crítico. Naturalmente todo grafo  $G$  admite um subgrafo  $\chi(G)$ -crítico. Além disso, todo grafo  $k$ -crítico é conexo. O teorema seguinte traz alguma informação sobre esta classe de grafos.

### Teorema 2.12

Se  $G(V, E)$  é  $k$ -crítico então  $\text{grau}(v) \geq k - 1$ , para todo  $v \in V$ .

**Prova** Suponha que  $G$  seja  $k$ -crítico e  $\text{grau}(v) < k - 1$ , para algum  $v \in V$ . Como  $G$  é  $k$ -crítico,  $G - v$  é  $(k - 1)$ -colorível. Seja  $V_1, \dots, V_{k-1}$  uma  $(k - 1)$ -coloração de  $G - v$ . Isto é,  $V_i$  representa o conjunto de vértices de  $G$  com a cor  $i$ ,  $1 \leq i \leq k - 1$ . Como  $\text{grau}(v) < k - 1$ , existe  $V_j$  tal que todo  $w \in V_j$  é não adjacente a  $v$ . Então  $V_1, \dots, V_j \cup \{v\}, \dots, V_{k-1}$  é uma  $(k - l)$ -coloração de  $G$ , uma contradição. ■

O estudo de coloração foi iniciado com o *problema das quatro cores*, conforme mencionado na Seção 1.2. Seja  $M$  uma região do plano. Um *mapa* é um partitionamento de  $M$  em um número finito de sub-regiões, as quais são, pois, delimitadas por linhas. Duas sub-regiões são *adjacentes* quando possuírem uma linha em comum. Uma *coloração* de  $M$  é uma atribuição de alguma cor a cada região de  $M$ , de modo que regiões adjacentes possuam cores diferentes. Um mapa  $M$  é  *$k$ -colorível* quando existir uma coloração de  $M$  que utiliza  $k$  cores. Observe que existe uma correspondência entre coloração de grafos  $G$  e de mapas  $M$ . De fato, defina  $G$  de modo a possuir um vértice para cada região de  $M$ , sendo um par de vértices adjacente quando as respectivas regiões o forem. Então é imediato observar que colorir o mapa  $M$  é equivalente a colorir o grafo  $G$ . Por outro lado, observe que  $G$  é necessariamente planar, pela própria definição de  $M$ . Então o problema de coloração de mapas é equivalente ao de coloração de grafos planares. Por exemplo, as Figuras 2.31(a) e (b) ilustram respectivamente uma 3-coloração de um mapa e seu grafo planar correspondente. O problema das quatro cores consiste, pois, em provar que todo grafo planar é 4-colorível.

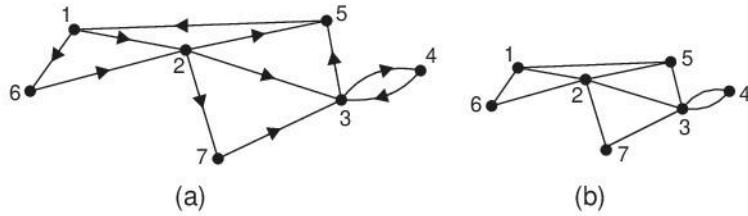


Figura 2.32: Um digrafo e seu grafo subjacente

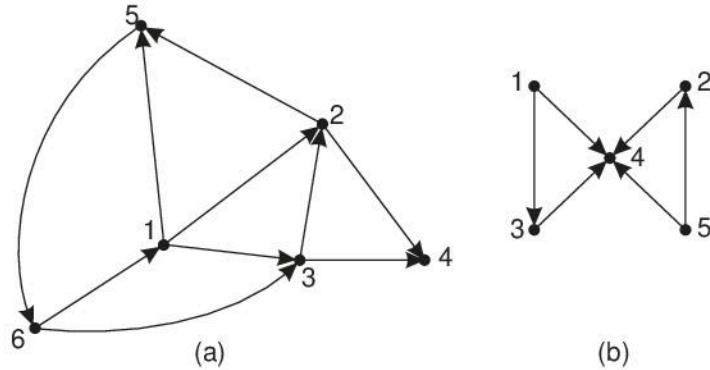


Figura 2.33: Digrafos unilateralmente e fracamente conexos

## 2.8 Grafos Direcionados

Os grafos examinados até o momento são também denominados *não direcionados* (Seção 2.2). Isto porque suas arestas  $(v, w)$  não são direcionadas. Assim, se  $(v, w)$  é aresta de  $G(V, E)$  então tanto  $v$  é adjacente a  $w$  como  $w$  é adjacente a  $v$ . Em contrapartida, um *grafo direcionado (digrafo)*  $D(V, E)$  é um conjunto finito não vazio  $V$  (*os vértices*), e um conjunto  $E$  (*as arestas*) de pares ordenados de vértices distintos. Portanto, num digrafo cada aresta  $(v, w)$  possui uma única direção de  $v$  para  $w$ . Diz-se também que  $(v, w)$  é *divergente* de  $v$  e *convergente* a  $w$ . Boa parte da nomenclatura e dos conceitos é análoga nos dois casos. Assim sendo, definem-se, por exemplo, caminho simples, trajeto, ciclo e ciclo simples de forma análoga às definições de grafos. Em particular, utilizamos o termo *subdigrafo* com o significado de subgrafo direcionado. Contudo, ao contrário dos grafos, os digrafos podem possuir ciclos de comprimento 2, no caso em que ambos  $(v, w)$  e  $(w, v)$  são arestas do digrafo. A Figura 2.32(a) ilustra um digrafo. Observe que entre os vértices 3 e 4 há um ciclo do comprimento 2. Os caminhos e ciclos em um digrafo devem obedecer ao direcionamento das arestas. Assim, na Figura 2.32(a), a sequência de vértices 2, 7, 3, 5 é um caminho de 2 para 5, enquanto que a 2, 7, 5 não o é.

Seja  $D(V, E)$  um digrafo e um vértice  $v \in V$ . O *grau de entrada* de  $v$  é o número de arestas convergentes a  $v$ . O *grau de saída* de  $v$  é o número de arestas divergentes de  $v$ . O vértice 3 da Figura 2.32(a) possui grau de entrada 3 e de saída 2. Uma *fonte* é um vértice com grau de entrada nulo, enquanto que um *sumidouro* (ou *poço*) é um com grau de saída nulo. Por exemplo, o vértice 1 da Figura 2.33(b) é uma fonte e o vértice 4 da mesma figura, um sumidouro.

Se forem retiradas as direções das arestas de um digrafo  $D$  obtém-se um multigrafo não direcionado, chamado *grafo subjacente* a  $D$ . Por exemplo, o multigrafo da Figura 2.32(b) é o subjacente ao digrafo 2.32(a).

Um digrafo  $D(V, E)$  é *fortemente conexo* quando para todo par de vértices  $v, w \in V$  existir um caminho em  $D$  de  $v$  para  $w$ , e também de  $w$  para  $v$ . Se ao menos um desses caminhos existir para todo  $v, w \in V$ , então  $D$  é *unilateralmente conexo*. Um digrafo é chamado *fracamente conexo* ou *desconexo*, conforme seu grafo subjacente seja conexo ou desconexo, respectivamente. Observe que se um digrafo é fortemente conexo então também é unilateralmente e fracamente conexo. Como exemplo, os digrafos das Figuras 2.32(a), 2.33(a) e 2.33(b) são respectivamente forte, unilateral e fracamente conexos. Uma *componente fortemente conexa* é um subdigrafo maximal de  $D$ , que seja fortemente conexo. Se existir algum caminho em  $D$  de um vértice  $v$  para um vértice  $w$ ,

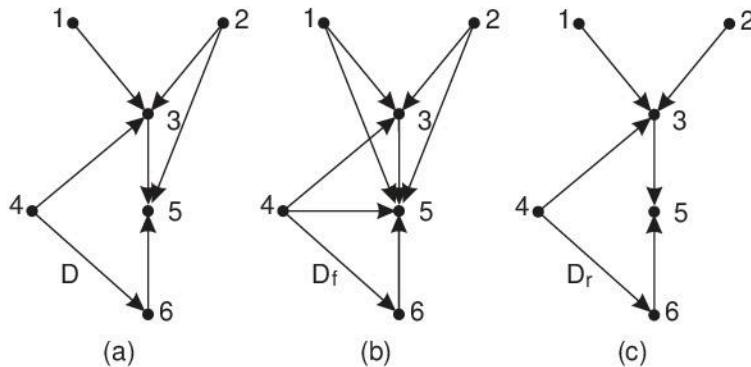


Figura 2.34: Um digrafo, seu fecho e redução transitivos

então diz-se que  $v$  alcança ou atinge  $w$ , sendo este alcançável ou atingível de  $v$ . Se  $v$  alcançar todos os vértices de  $D$  então  $v$  é chamado raiz do digrafo.

Um digrafo é acíclico quando não possui ciclos. Observe que o grafo subjacente a um digrafo acíclico não é necessariamente acíclico. Mas é acíclico um digrafo cujo grafo subjacente também o seja.

### Lema 2.8

*Todo digrafo acíclico possui (pelo menos) uma fonte e um sumidouro.*

Seja  $D(V, E)$  um digrafo acíclico. Denomina-se fecho transitivo de  $D$  ao maior digrafo  $D_f(V, E_f)$  que preserva a alcançabilidade de  $D$ . Isto é, para todo  $v, w \in V$ , se  $v$  alcança  $w$  em  $D$  então  $(v, w) \in E_f$ . Analogamente, a redução transitiva de  $D$  é o menor digrafo  $D_r(V, E_r)$  que preserva a alcançabilidade de  $D$ . Isto é, se  $(v, w) \in E_r$  então  $v$  não alcança  $w$  em  $D - (v, w)$ . A redução transitiva é também conhecida como diagrama de Hasse. Os digrafos das Figuras 2.34(b) e (c) são respectivamente o fecho e redução transitivos daquele da 2.34(a).

Um conjunto parcialmente ordenado ou ordenação parcial  $(S, \leq)$  é um conjunto não vazio  $S$  e uma relação binária  $\leq$  em  $S$ , satisfazendo às seguintes propriedades:

- (i)  $s_1 \leq s_1$  para  $s_1 \in S$  ( $\leq$  é reflexiva).
- (ii)  $s_1 \leq s_2$  e  $s_2 \leq s_1 \Rightarrow s_1 = s_2$ , para  $s_1, s_2 \in S$  ( $\leq$  é anti-simétrica).
- (iii)  $s_1 \leq s_2$  e  $s_2 \leq s_3 \Rightarrow s_1 \leq s_3$ , para  $s_1, s_2, s_3 \in S$  ( $\leq$  é transitiva).

Um conjunto parcialmente ordenado  $(S, \leq)$  pode ser também caracterizado pelo par  $(S, <)$  onde  $<$  é uma relação binária em  $S$  definida por  $s_1 < s_2 \Leftrightarrow s_1 \leq s_2$  e  $s_1 \neq s_2$ . A relação  $<$  satisfaz às relações abaixo, as quais não são independentes entre si.

- (i)  $s_1 \not< s_1$  para  $s_1 \in S$  ( $<$  é irreflexiva).
- (ii)  $s_1 < s_2 \Rightarrow s_2 \not< s_1$ , para  $s_1, s_2 \in S$  ( $<$  é assimétrica).
- (iii)  $s_1 < s_2$  e  $s_2 < s_3 \Rightarrow s_1 < s_3$ , para  $s_1, s_2, s_3 \in S$  ( $<$  é transitiva).

Seja  $D_f(V, E_f)$  o fecho transitivo de um digrafo acíclico  $D(V, E)$ . Pode-se constatar que  $E_f$  é uma relação irreflexiva, assimétrica e transitiva. Consequentemente, fechos transitivos de digrafos acíclicos e conjuntos parcialmente ordenados são conceitos equivalentes. Isto é,  $P(S, <)$  caracteriza um conjunto parcialmente ordenado se e somente se o digrafo  $D_f(S, <)$  for o fecho transitivo de algum digrafo acíclico  $D$ . Diz-se então que  $D$  induz o conjunto parcialmente ordenado  $P(S, <)$ . Observe que  $s_1 < s_2$  em  $P$  se e somente se existir um caminho em  $D$  de  $s_1$  para  $s_2$ . Utiliza-se então a notação  $v < w$  para indicar que  $v$  alcança  $w$  no digrafo acíclico  $D(V, E)$ , sendo  $v, w \in V$ .

Finalmente, uma árvore direcionada enraizada é um digrafo  $D$  tal que exatamente um vértice  $r$  possui grau de entrada nulo, enquanto para todos os demais vértices o grau de entrada é igual a um. É imediato observar que  $D$  é acíclico com exatamente uma raiz  $r$ . Além disso, o grafo subjacente  $T$  de  $D$  é uma árvore. Por isso é usual aplicar a nomenclatura de árvores a esta classe de digrafos. Ver Figura 2.35.

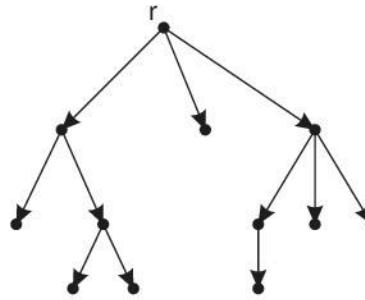


Figura 2.35: Uma árvore direcionada enraizada

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(a)

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(b)

Figura 2.36: Matrizes de adjacências

## 2.9 Representação de Grafos

Nesta seção examinam-se algumas representações de grafos, adequadas ao uso em computador. Ou seja, estruturas que (i) correspondam univocamente a um grafo dado e (ii) possam ser armazenadas e manipuladas sem dificuldade, em um computador. Observe que a representação geométrica, por exemplo, não satisfaz à condição (ii). Dentre os vários tipos de representações adequadas ao computador, ressaltam-se as *representações matriciais* e as *por listas*. A seguir, são examinadas algumas dessas.

Dado um grafo  $G(V, E)$  a *matriz de adjacências*  $R = (r_{ij})$  é uma matriz  $n \times n$  tal que:

$$r_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E$$

$$r_{ij} = 0 \text{ caso contrário.}$$

Ou seja,  $r_{ij} = 1$  quando os vértices  $v_i, v_j$  forem adjacentes e  $r_{ij} = 0$ , caso contrário. É imediato verificar que a matriz de adjacências representa um grafo sem ambiguidade. Além disso, é de simples manipulação em computador. Algumas propriedades da matriz  $R$  são imediatas. Por exemplo,  $R$  é simétrica para um grafo não direcionado. Além disso, o número de 1's é igual a  $2m$ , pois cada aresta  $(v_i, v_j)$  dá origem a dois 1's em  $A$ ,  $r_{ij}$  e  $r_{ji}$ .

Observe que uma matriz de adjacências caracteriza univocamente um grafo. Contudo a um mesmo grafo  $G$  podem corresponder várias matrizes diferentes. De fato, se  $R_1$  é matriz de adjacências de  $G$  e  $R_2$  é outra matriz obtida por alguma permutação de linhas e colunas de  $R_1$ , então  $R_2$  também é matriz de adjacências de  $G$ . Ou seja, para construir uma matriz de adjacências de  $G(V, E)$  é necessário arbitrar uma certa permutação  $v_1, v_2, \dots, v_n$  para os vértices de  $V$ . Naturalmente, permutações diferentes podem conduzir a matrizes diferentes. Observe que o problema de isomorfismo de grafos pode ser então formulado nos seguintes termos: “sendo  $R_1$  e  $R_2$  duas matrizes de adjacências dadas, representam  $R_1$  e  $R_2$  o mesmo grafo?”. As Figuras 2.36(a) e (b) ilustram duas matrizes de adjacências do grafo da Figura 2.1, correspondentes às permutações dos vértices 1, 2, 3, 4, 5, 6 e 5, 4, 1, 3, 2, 6, respectivamente.

A matriz de adjacências pode ser também definida para digrafos. Basta tomar  $r_{ij} = 1 \Leftrightarrow (v_i, v_j)$  for aresta divergente de  $v_i$  e convergente a  $v_j$ , e  $r_{ij} = 0$ , caso contrário. Naturalmente, a matriz não é mais necessariamente simétrica e o número de 1's é exatamente igual a  $m$ .

Em relação ao espaço necessário ao armazenamento da matriz, em qualquer caso existem  $n^2$  informações binárias, o que significa um espaço  $O(n^2)$ . Ou seja, um espaço não linear com o tamanho do grafo (número de

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figura 2.37: Matriz de incidências

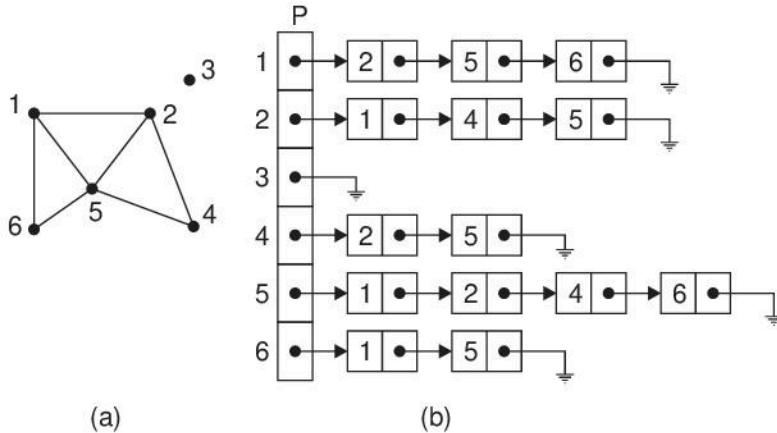


Figura 2.38: Estrutura de adjacência

vértices e arestas), no caso em que este for esparsa (isto é,  $m = O(n)$ ). Esta é a principal desvantagem dessa representação.

Uma outra representação matricial para o grafo  $G(V, E)$  é a *matriz de incidências*  $n \times mB = (b_{ij})$  assim definida:

$b_{ij} = 1 \Leftrightarrow$  vértice  $v_i$  e aresta  $e_j$  forem incidentes

$b_{ij} = 0$  caso contrário.

Ou seja, arbitram-se permutações para os vértices  $v_1, \dots, v_n$  e para as arestas  $e_1, \dots, e_m$  de  $G$ . Então  $b_{ij} = 1$  precisamente quando o vértice  $v_i$  for um extremo da aresta  $e_j$ , e  $b_{ij} = 0$ , caso contrário. Também nesse caso é imediato verificar que a matriz de incidências representa univocamente um grafo, mas este último pode ser representado, em geral, por várias matrizes de incidências diferentes. Observe que cada coluna de  $B$  tem exatamente dois 1's. A Figura 2.37 ilustra a matriz de incidências do grafo da Figura 2.1, correspondente às permutações de vértices 1, 2, 3, 4, 5, 6 e de arestas (1, 2), (1, 3), (1, 6), (1, 5), (2, 3), (2, 6), (3, 5), (3, 6), (5, 4), (5, 6), (4, 6), (3, 4).

Existem várias representações de grafos por listas. Dentre essas, é muito comum a *estrutura de adjacências*. Seja  $G(V, E)$  um grafo. A estrutura de *adjacências*,  $A$  de  $G$  é um conjunto de  $n$  listas  $A(v)$ , uma para cada  $v \in V$ . Cada lista  $A(v)$  é denominada *lista de adjacências* do vértice  $v$ , e contém os vértices  $w$  adjacentes a  $v$  em  $G$ . Ou seja,  $A(v) = \{w | (v, w) \in E\}$ . A Figura 2.38(b) ilustra as listas de adjacências do grafo da Figura 2.38(a).

Observe que cada aresta  $(v, w)$  dá origem a duas entradas na estrutura de adjacências, correspondentes a  $v \in A(w)$  e  $w \in A(v)$ . Logo, a estrutura  $A$  consiste em  $n$  listas com um total de  $2m$  elementos. A cada elemento  $w$  de uma lista de adjacências  $A(v)$  associa-se um ponteiro, o qual informa o próximo elemento, se houver, após  $w$  em  $A(v)$ . Além disso, é necessária também a utilização de um vetor  $p$ , de tamanho  $n$ , tal que  $p(v)$  indique o ponteiro inicial da lista  $A(v)$ . Assim sendo, se  $A(v)$  for vazia,  $p(v) = \emptyset$ . Pode-se concluir então que o espaço utilizado por uma estrutura da adjacências é  $O(n + m)$ , ou seja, linear com o tamanho de  $G$ . Esta é uma das razões pelas quais a estrutura de adjacências talvez seja a representação mais comumente encontrada nas implementações dos algoritmos em grafos.

A estrutura da adjacência pode ser interpretada como uma matriz de adjacências representada sob a forma de matriz esparsa, isto é, na qual os zeros não estão presentes. Nesse caso,  $v_j \in A(v_i)$  corresponderia a afirmar que  $r_{ij} = 1$  na matriz de adjacências. Finalmente observe que a estrutura da adjacência pode ser definida para digrafos de forma análoga. Isto é, para um digrafo  $D(V, E)$  define-se uma lista de adjacências  $A(v)$ , para cada  $v \in V$ . A lista  $A(v)$  é formada pelos vértices divergentes de  $v$ , isto é,  $A(v) = \{w | (v, w) \in E\}$ . A estrutura de adjacências, nesse caso, contém  $m$  elementos. É imediato verificar que o espaço requerido pela estrutura  $A$  para representar o digrafo  $D$  é também linear com o tamanho de  $D$ .

## 2.10 Implementações em Python: Operações Básicas

É comum que algoritmos em grafos sejam apresentados supondo-se que as operações básicas sobre grafos estejam previamente definidas por algoritmos mais elementares. As implementações dos diversos algoritmos deste livro nos quais grafos são utilizados necessitam da implementação das seguintes operações básicas, tanto na representação de matriz de adjacências, quanto naquela de lista de adjacências:

- (i) Definição do grafo
- (ii) Adição de arestas
- (iii) Determinação de vizinhos de dado vértice
- (iv) Teste de vizinhança entre par de vértices
- (v) Remoção de arestas

A próxima implementação descreve a classe `Grafo`, a ser usada como base em ambas representações (base no sentido da orientação a objetos, ou seja, uma classe que pode derivar outras específicas). O procedimento `__init__` é a inicialização da estrutura de dados do grafo, que basicamente armazena o tipo de grafo (direcionado ou não). Nesta implementação, é necessário executar o procedimento `DefinirN` ainda como parte da construção do grafo, definindo o seu número de vértices. As implementações específicas de grafos, a saber `GrafoMatrizAdj` e `GrafoListaAdj` definem versões particulares deste procedimento, alocando respectivamente a matriz de adjacências e a lista de adjacências, ambas inicialmente sem representar arestas (grafo vazio).

É comum que os algoritmos necessitem iterar sobre o conjunto de vértices ou de arestas do grafo e, para tal propósito, há as funções `V` e `E`. O retorno da função `V` é um iterador sobre o conjunto de vértices que, por padrão, é sempre uma sequência de inteiros entre 1 e  $n$ . Já o retorno da função `E` é um iterador sobre o conjunto de pares de vértices representando as arestas. Alternativamente, no caso de lista de adjacências e se o parâmetro `IterarSobreNo=True`, o primeiro valor de cada par representando a aresta  $(v, w)$  é o vértice  $v$  e o segundo consiste no nó da lista de vizinhos de  $v$  que representa  $w$ .

```
1 class Grafo(object):
2     """
3     Classe base para as classes GrafoListaAdj e GrafoMatrizAdj
4     """
5     def __init__(self, orientado = False):
6         """
7             Grafo se orientado=False ou Digrafo se orientado=True.
8             """
9         self.n, self.m, self.orientado = None, None, orientado
10
11    def DefinirN(self, n):
12        """
13            Define o número n de vértices.
14            """
15        self.n, self.m = n, 0
16
17    def V(self):
```

```

18     """
19     Retorna a lista de vértices.
20     """
21     for i in range(1, self.n+1):
22         yield i
23
24 def E(self, IterarSobreNo=False):
25     """
26     Retorna lista de arestas uv, onde u é um inteiro e v é um inteiro se o grafo é ↓
27     GrafoMatrizAdj
28     ou IterarSobreNo=False; v é GrafoListaAdj.NoAresta, caso contrário.
29     """
30     for v in self.V():
31         for w in self.N(v, Tipo = "+" if self.orientado else "*", ↓
32             IterarSobreNo=IterarSobreNo):
33             enumerar = True
34             if not self.orientado:
35                 wint = w if isinstance(w, int) else w.Viz
36                 enumerar = v < wint
37             if enumerar:
38                 yield (v, w)

```

As operações sobre grafos para matriz de adjacências são implementadas como a seguir. Algumas dessas são diretas e decorrem da definição da matriz de adjacências. A função N retorna um iterador sobre os vizinhos de dado vértice  $v$  e inclui o próprio vértice  $v$  se o parâmetro Fechada=True, indicando que se trata da vizinhança fechada de  $v$ . O parâmetro Tipo é considerado no caso de grafos direcionados, que indica se a vizinhança a ser retornada é a de entrada (Tipo="-"), de saída (Tipo="+") ou qualquer vizinho (Tipo="\*").

```

1 class GrafoMatrizAdj(Grafo):
2
3     def DefinirN(self, n):
4         """
5         Define o número n de vértices.
6         """
7         super(GrafoMatrizAdj, self).DefinirN(n)
8         self.M = [None]*(self.n+1)
9         for i in range(1, self.n+1):
10            self.M[i] = [0]*(self.n+1)
11
12     def RemoverAresta(self, u, v):
13         """
14         Remove a aresta uv.
15         """
16         self.M[u][v] = 0
17         if not self.orientado:
18             self.M[v][u] = 0
19             self.m = self.m-1
20
21     def AdicionarAresta(self, u, v):
22         """
23         Adiciona aresta uv.
24         """
25         self.M[u][v] = 1
26         if not self.orientado:
27             self.M[v][u] = 1
28             self.m = self.m+1
29

```

```

30     def SaoAdj(self, u, v):
31         """
32             Retorna True se e somente se uv é uma aresta.
33             """
34         return self.M[u][v] == 1
35
36     def N(self, v, Tipo = "*", Fechada=False, IterarSobreNo=False):
37         """
38             Retorna lista de vértices vizinhos do vértice v. Se Fechada=True, o próprio v é ↓
39             incluído na lista.
40             Tipo="*" significa listar todas as arestas incidentes em v. Se G é orientado,
41             Tipo="+" (resp. "-") significa listar apenas as arestas de saída (resp. entrada) de v.
42             IterarSobreNo não tem uso para Matriz de Adjacências.
43             """
44         if Fechada:
45             yield v
46         w = 1
47         t = "+" if Tipo == "*" and self.orientado else Tipo
48         while w <= self.n:
49             if t == "+":
50                 orig, dest, viz = v, w, w
51             else:
52                 orig, dest, viz = w, v, w
53             if self.SaoAdj(orig, dest):
54                 yield w
55             w = w+1
56             if w > self.n and t == "+" and Tipo == "*":
57                 t, w = "-", 1

```

Tais operações sobre grafos para lista de adjacências são implementadas conforme indicado.

A lista encadeada de vizinhos é implementada como uma lista simplesmente ou duplamente encadeada, dependendo do valor informado no parâmetro VizinhancaDuplamenteLigada passado no procedimento DefinirN. As listas duplamente encadeadas possibilitam a remoção de uma aresta em tempo constante, enquanto as listas simplesmente encadeadas economizam a memória de dois ponteiros por aresta. De todo modo, a lista encadeada é implementada com nó-cabeça.

Cada aresta  $(u, v)$  do grafo é representada por três objetos em memória: (i) um objeto do tipo NoAresta, representando o nó que corresponde ao vértice  $v$  na lista de vizinhos de  $u$  (com a informação de que é um vizinho de saída, caso seja um digrafo); (ii) um objeto do tipo NoAresta, representando o nó que corresponde ao vértice  $u$  na lista de vizinhos de  $v$  (com a informação de que é um vizinho de entrada, caso seja um digrafo); (iii) um objeto do tipo Aresta, com os atributos da aresta (tipicamente peso, custo, etc.), além de servir de ligação entre os dois objetos NoAresta anteriores, como será detalhado adiante.

Cada nó da lista de adjacências, representado por um objeto do tipo NoAresta, possui os seguintes atributos: (i) Viz, com o vértice vizinho correspondente ao nó; (ii) Prox, com o próximo nó da lista encadeada; (iii) Ant, caso a lista seja duplamente encadeada, com o nó anterior da lista encadeada (sempre existirá um nó anterior, visto a presença de nó-cabeça); (iv) Tipo, para o caso de digrafos, com o valor "+" (resp. "-") para indicar que se trata de um vizinho de saída (resp. entrada); e finalmente (v) o objeto Aresta, que contém os atributos da aresta.

Cada nó correspondente a uma aresta, representado por um objeto do tipo Aresta, possui os seguintes atributos: (i) v1 e v2, para indicar que representa a aresta  $(v_1, v_2)$ ; (ii) No1 e No2, apontadores para respectivamente o nó da lista encadeada de v1 referente a v2 e para o nó da lista encadeada de v2 referente a v1; (iii) outros atributos definidos no contexto do problema, como custo, distância, fluxo, marcações, etc.

A Figura 2.39 representa a lista de adjacências criada para representar o grafo da Figura 2.10(b). Na figura, apenas os ponteiros entre os objetos NoAresta e Aresta correspondentes à aresta  $(1, 4)$  foram representados por simplicidade.

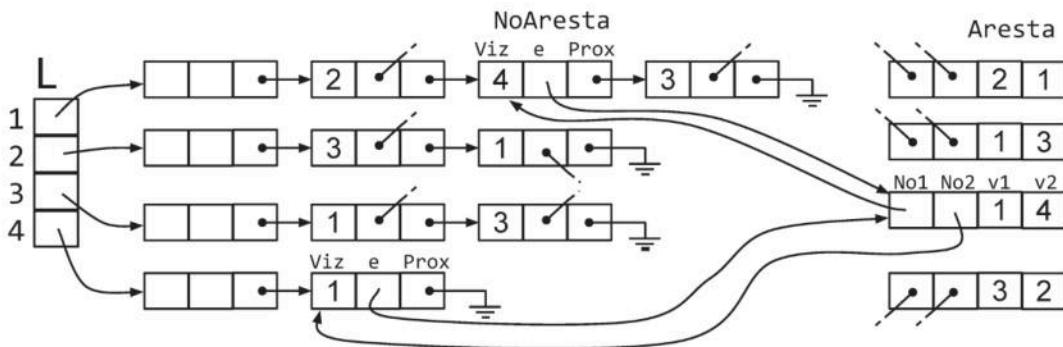


Figura 2.39: Lista de adjacência

```

1 class GrafoListaAdj(Grafo):
2     class NoAresta(object):
3         """
4             Objeto nó da lista de adjacência.
5             Atributos:
6                 - Viz (vizinho)
7                 - e (Aresta)
8                 - Tipo (+/-)
9                 - Prox (próxima aresta na lista de adjacência)
10                - Ant (aresta anterior na lista de adjacência (se a lista for duplamente encadeada))
11            """
12
13     def __init__(self):
14         self.Viz = None
15         self.e = None
16         self.Prox = None
17
18     class Aresta(object):
19         """
20             Objeto único para representar a aresta.
21             Atributos:
22                 - v1, No1 (um dos vértices desta aresta e seu respectivo nó, isto é, v1 == No1.Viz)
23                 - v2, No2 (análogo em relação ao segundo vértice)
24         """
25     def __init__(self):
26         self.v1, self.No1 = None, None
27         self.v2, self.No2 = None, None
28
29     def DefinirN(self, n, VizinhancaDuplamenteLigada=False):
30         """
31             Define o número n de vértices.
32             Se VizinhancaDuplamenteLigada=True, a lista encadeada dos vizinhos de um vértice é ↓
33             duplamente ligada (permitindo remoção de arestas de tempo constante).
34         """
35         super(GrafoListaAdj, self).DefinirN(n)
36         self.L = [None] * (self.n + 1)
37         for i in range(1, self.n + 1):
38             self.L[i] = GrafoListaAdj.NoAresta() #nó cabeça
39             self.VizinhancaDuplamenteLigada = VizinhancaDuplamenteLigada
40
41     def AdicionarAresta(self, u, v):
42         """
43             Adiciona uma nova aresta entre os vértices u e v.
44             Se a aresta já existe, ela é removida.
45             Se a lista de adjacência é duplamente ligada, a remoção é feita em tempo constante.
46         """
47         if u > self.n or v > self.n:
48             raise ValueError("Vértices inválidos")
49
50         if self.VizinhancaDuplamenteLigada:
51             self._removerAresta(u, v)
52
53         aresta = self._criarAresta(u, v)
54
55         self._adicionarVizinho(u, v, aresta)
56         self._adicionarVizinho(v, u, aresta)
57
58         self._atualizarProximos(aresta)
59
60     def _criarAresta(self, u, v):
61         """
62             Cria uma nova aresta entre os vértices u e v.
63         """
64         aresta = self.Aresta()
65         aresta.v1 = u
66         aresta.v2 = v
67         aresta.No1 = self.NoAresta(u)
68         aresta.No2 = self.NoAresta(v)
69
70         return aresta
71
72     def _removerAresta(self, u, v):
73         """
74             Remove a aresta entre os vértices u e v.
75             Se a aresta não existir, não há efeitos colaterais.
76         """
77         aresta = self._acharAresta(u, v)
78
79         if aresta is None:
80             return
81
82         self._removerVizinho(u, v, aresta)
83         self._removerVizinho(v, u, aresta)
84
85         self._atualizarProximos(aresta)
86
87     def _acharAresta(self, u, v):
88         """
89             Busca a aresta entre os vértices u e v.
90             Se a aresta não existir, retorna None.
91         """
92         aresta = None
93
94         for i in range(1, self.n + 1):
95             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
96                 aresta = self.L[i].e
97                 break
98
99         return aresta
100
101     def _adicionarVizinho(self, u, v, aresta):
102         """
103             Adiciona o vértice v como vizinho do vértice u.
104             A aresta é adicionada à lista de vizinhos do vértice u.
105         """
106         self.L[u].Viz = v
107         self.L[u].e = aresta
108
109         aresta.No1 = self.NoAresta(u)
110         aresta.No2 = self.NoAresta(v)
111
112     def _removerVizinho(self, u, v, aresta):
113         """
114             Remove o vértice v como vizinho do vértice u.
115             A aresta é removida da lista de vizinhos do vértice u.
116         """
117         self.L[u].Viz = None
118         self.L[u].e = None
119
120         aresta.No1 = None
121         aresta.No2 = None
122
123     def _atualizarProximos(self, aresta):
124         """
125             Atualiza os próximos apontamentos para a aresta.
126             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
127         """
128         if self.VizinhancaDuplamenteLigada:
129             prox = aresta.No1.Prox
130             aresta.No1.Prox = aresta.No2
131             aresta.No2.Ant = prox
132
133             prox = aresta.No2.Prox
134             aresta.No2.Prox = aresta.No1
135             aresta.No1.Ant = prox
136
137         else:
138             aresta.No1.Prox = aresta.No2
139             aresta.No2.Ant = aresta.No1
140
141             aresta.No2.Prox = aresta.No1
142             aresta.No1.Ant = aresta.No2
143
144     def _criarNoAresta(self, u):
145         """
146             Cria um novo nó para a lista de adjacência do vértice u.
147         """
148         return self.NoAresta(u)
149
150     def _criarAresta(self, u, v):
151         """
152             Cria uma nova aresta entre os vértices u e v.
153         """
154         aresta = self.Aresta()
155         aresta.v1 = u
156         aresta.v2 = v
157         aresta.No1 = self.NoAresta(u)
158         aresta.No2 = self.NoAresta(v)
159
160         return aresta
161
162     def _removerAresta(self, u, v):
163         """
164             Remove a aresta entre os vértices u e v.
165             Se a aresta não existir, não há efeitos colaterais.
166         """
167         aresta = self._acharAresta(u, v)
168
169         if aresta is None:
170             return
171
172         self._removerVizinho(u, v, aresta)
173         self._removerVizinho(v, u, aresta)
174
175         self._atualizarProximos(aresta)
176
177     def _acharAresta(self, u, v):
178         """
179             Busca a aresta entre os vértices u e v.
180             Se a aresta não existir, retorna None.
181         """
182         aresta = None
183
184         for i in range(1, self.n + 1):
185             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
186                 aresta = self.L[i].e
187                 break
188
189         return aresta
190
191     def _adicionarVizinho(self, u, v, aresta):
192         """
193             Adiciona o vértice v como vizinho do vértice u.
194             A aresta é adicionada à lista de vizinhos do vértice u.
195         """
196         self.L[u].Viz = v
197         self.L[u].e = aresta
198
199         aresta.No1 = self.NoAresta(u)
200         aresta.No2 = self.NoAresta(v)
201
202     def _removerVizinho(self, u, v, aresta):
203         """
204             Remove o vértice v como vizinho do vértice u.
205             A aresta é removida da lista de vizinhos do vértice u.
206         """
207         self.L[u].Viz = None
208         self.L[u].e = None
209
210         aresta.No1 = None
211         aresta.No2 = None
212
213     def _atualizarProximos(self, aresta):
214         """
215             Atualiza os próximos apontamentos para a aresta.
216             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
217         """
218         if self.VizinhancaDuplamenteLigada:
219             prox = aresta.No1.Prox
220             aresta.No1.Prox = aresta.No2
221             aresta.No2.Ant = prox
222
223             prox = aresta.No2.Prox
224             aresta.No2.Prox = aresta.No1
225             aresta.No1.Ant = prox
226
227         else:
228             aresta.No1.Prox = aresta.No2
229             aresta.No2.Ant = aresta.No1
230
231             aresta.No2.Prox = aresta.No1
232             aresta.No1.Ant = aresta.No2
233
234     def _criarNoAresta(self, u):
235         """
236             Cria um novo nó para a lista de adjacência do vértice u.
237         """
238         return self.NoAresta(u)
239
240     def _criarAresta(self, u, v):
241         """
242             Cria uma nova aresta entre os vértices u e v.
243         """
244         aresta = self.Aresta()
245         aresta.v1 = u
246         aresta.v2 = v
247         aresta.No1 = self.NoAresta(u)
248         aresta.No2 = self.NoAresta(v)
249
250         return aresta
251
252     def _removerAresta(self, u, v):
253         """
254             Remove a aresta entre os vértices u e v.
255             Se a aresta não existir, não há efeitos colaterais.
256         """
257         aresta = self._acharAresta(u, v)
258
259         if aresta is None:
260             return
261
262         self._removerVizinho(u, v, aresta)
263         self._removerVizinho(v, u, aresta)
264
265         self._atualizarProximos(aresta)
266
267     def _acharAresta(self, u, v):
268         """
269             Busca a aresta entre os vértices u e v.
270             Se a aresta não existir, retorna None.
271         """
272         aresta = None
273
274         for i in range(1, self.n + 1):
275             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
276                 aresta = self.L[i].e
277                 break
278
279         return aresta
280
281     def _adicionarVizinho(self, u, v, aresta):
282         """
283             Adiciona o vértice v como vizinho do vértice u.
284             A aresta é adicionada à lista de vizinhos do vértice u.
285         """
286         self.L[u].Viz = v
287         self.L[u].e = aresta
288
289         aresta.No1 = self.NoAresta(u)
290         aresta.No2 = self.NoAresta(v)
291
292     def _removerVizinho(self, u, v, aresta):
293         """
294             Remove o vértice v como vizinho do vértice u.
295             A aresta é removida da lista de vizinhos do vértice u.
296         """
297         self.L[u].Viz = None
298         self.L[u].e = None
299
300         aresta.No1 = None
301         aresta.No2 = None
302
303     def _atualizarProximos(self, aresta):
304         """
305             Atualiza os próximos apontamentos para a aresta.
306             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
307         """
308         if self.VizinhancaDuplamenteLigada:
309             prox = aresta.No1.Prox
310             aresta.No1.Prox = aresta.No2
311             aresta.No2.Ant = prox
312
313             prox = aresta.No2.Prox
314             aresta.No2.Prox = aresta.No1
315             aresta.No1.Ant = prox
316
317         else:
318             aresta.No1.Prox = aresta.No2
319             aresta.No2.Ant = aresta.No1
320
321             aresta.No2.Prox = aresta.No1
322             aresta.No1.Ant = aresta.No2
323
324     def _criarNoAresta(self, u):
325         """
326             Cria um novo nó para a lista de adjacência do vértice u.
327         """
328         return self.NoAresta(u)
329
330     def _criarAresta(self, u, v):
331         """
332             Cria uma nova aresta entre os vértices u e v.
333         """
334         aresta = self.Aresta()
335         aresta.v1 = u
336         aresta.v2 = v
337         aresta.No1 = self.NoAresta(u)
338         aresta.No2 = self.NoAresta(v)
339
340         return aresta
341
342     def _removerAresta(self, u, v):
343         """
344             Remove a aresta entre os vértices u e v.
345             Se a aresta não existir, não há efeitos colaterais.
346         """
347         aresta = self._acharAresta(u, v)
348
349         if aresta is None:
350             return
351
352         self._removerVizinho(u, v, aresta)
353         self._removerVizinho(v, u, aresta)
354
355         self._atualizarProximos(aresta)
356
357     def _acharAresta(self, u, v):
358         """
359             Busca a aresta entre os vértices u e v.
360             Se a aresta não existir, retorna None.
361         """
362         aresta = None
363
364         for i in range(1, self.n + 1):
365             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
366                 aresta = self.L[i].e
367                 break
368
369         return aresta
370
371     def _adicionarVizinho(self, u, v, aresta):
372         """
373             Adiciona o vértice v como vizinho do vértice u.
374             A aresta é adicionada à lista de vizinhos do vértice u.
375         """
376         self.L[u].Viz = v
377         self.L[u].e = aresta
378
379         aresta.No1 = self.NoAresta(u)
380         aresta.No2 = self.NoAresta(v)
381
382     def _removerVizinho(self, u, v, aresta):
383         """
384             Remove o vértice v como vizinho do vértice u.
385             A aresta é removida da lista de vizinhos do vértice u.
386         """
387         self.L[u].Viz = None
388         self.L[u].e = None
389
390         aresta.No1 = None
391         aresta.No2 = None
392
393     def _atualizarProximos(self, aresta):
394         """
395             Atualiza os próximos apontamentos para a aresta.
396             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
397         """
398         if self.VizinhancaDuplamenteLigada:
399             prox = aresta.No1.Prox
400             aresta.No1.Prox = aresta.No2
401             aresta.No2.Ant = prox
402
403             prox = aresta.No2.Prox
404             aresta.No2.Prox = aresta.No1
405             aresta.No1.Ant = prox
406
407         else:
408             aresta.No1.Prox = aresta.No2
409             aresta.No2.Ant = aresta.No1
410
411             aresta.No2.Prox = aresta.No1
412             aresta.No1.Ant = aresta.No2
413
414     def _criarNoAresta(self, u):
415         """
416             Cria um novo nó para a lista de adjacência do vértice u.
417         """
418         return self.NoAresta(u)
419
420     def _criarAresta(self, u, v):
421         """
422             Cria uma nova aresta entre os vértices u e v.
423         """
424         aresta = self.Aresta()
425         aresta.v1 = u
426         aresta.v2 = v
427         aresta.No1 = self.NoAresta(u)
428         aresta.No2 = self.NoAresta(v)
429
430         return aresta
431
432     def _removerAresta(self, u, v):
433         """
434             Remove a aresta entre os vértices u e v.
435             Se a aresta não existir, não há efeitos colaterais.
436         """
437         aresta = self._acharAresta(u, v)
438
439         if aresta is None:
440             return
441
442         self._removerVizinho(u, v, aresta)
443         self._removerVizinho(v, u, aresta)
444
445         self._atualizarProximos(aresta)
446
447     def _acharAresta(self, u, v):
448         """
449             Busca a aresta entre os vértices u e v.
450             Se a aresta não existir, retorna None.
451         """
452         aresta = None
453
454         for i in range(1, self.n + 1):
455             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
456                 aresta = self.L[i].e
457                 break
458
459         return aresta
460
461     def _adicionarVizinho(self, u, v, aresta):
462         """
463             Adiciona o vértice v como vizinho do vértice u.
464             A aresta é adicionada à lista de vizinhos do vértice u.
465         """
466         self.L[u].Viz = v
467         self.L[u].e = aresta
468
469         aresta.No1 = self.NoAresta(u)
470         aresta.No2 = self.NoAresta(v)
471
472     def _removerVizinho(self, u, v, aresta):
473         """
474             Remove o vértice v como vizinho do vértice u.
475             A aresta é removida da lista de vizinhos do vértice u.
476         """
477         self.L[u].Viz = None
478         self.L[u].e = None
479
480         aresta.No1 = None
481         aresta.No2 = None
482
483     def _atualizarProximos(self, aresta):
484         """
485             Atualiza os próximos apontamentos para a aresta.
486             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
487         """
488         if self.VizinhancaDuplamenteLigada:
489             prox = aresta.No1.Prox
490             aresta.No1.Prox = aresta.No2
491             aresta.No2.Ant = prox
492
493             prox = aresta.No2.Prox
494             aresta.No2.Prox = aresta.No1
495             aresta.No1.Ant = prox
496
497         else:
498             aresta.No1.Prox = aresta.No2
499             aresta.No2.Ant = aresta.No1
500
501             aresta.No2.Prox = aresta.No1
502             aresta.No1.Ant = aresta.No2
503
504     def _criarNoAresta(self, u):
505         """
506             Cria um novo nó para a lista de adjacência do vértice u.
507         """
508         return self.NoAresta(u)
509
510     def _criarAresta(self, u, v):
511         """
512             Cria uma nova aresta entre os vértices u e v.
513         """
514         aresta = self.Aresta()
515         aresta.v1 = u
516         aresta.v2 = v
517         aresta.No1 = self.NoAresta(u)
518         aresta.No2 = self.NoAresta(v)
519
520         return aresta
521
522     def _removerAresta(self, u, v):
523         """
524             Remove a aresta entre os vértices u e v.
525             Se a aresta não existir, não há efeitos colaterais.
526         """
527         aresta = self._acharAresta(u, v)
528
529         if aresta is None:
530             return
531
532         self._removerVizinho(u, v, aresta)
533         self._removerVizinho(v, u, aresta)
534
535         self._atualizarProximos(aresta)
536
537     def _acharAresta(self, u, v):
538         """
539             Busca a aresta entre os vértices u e v.
540             Se a aresta não existir, retorna None.
541         """
542         aresta = None
543
544         for i in range(1, self.n + 1):
545             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
546                 aresta = self.L[i].e
547                 break
548
549         return aresta
550
551     def _adicionarVizinho(self, u, v, aresta):
552         """
553             Adiciona o vértice v como vizinho do vértice u.
554             A aresta é adicionada à lista de vizinhos do vértice u.
555         """
556         self.L[u].Viz = v
557         self.L[u].e = aresta
558
559         aresta.No1 = self.NoAresta(u)
560         aresta.No2 = self.NoAresta(v)
561
562     def _removerVizinho(self, u, v, aresta):
563         """
564             Remove o vértice v como vizinho do vértice u.
565             A aresta é removida da lista de vizinhos do vértice u.
566         """
567         self.L[u].Viz = None
568         self.L[u].e = None
569
570         aresta.No1 = None
571         aresta.No2 = None
572
573     def _atualizarProximos(self, aresta):
574         """
575             Atualiza os próximos apontamentos para a aresta.
576             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
577         """
578         if self.VizinhancaDuplamenteLigada:
579             prox = aresta.No1.Prox
580             aresta.No1.Prox = aresta.No2
581             aresta.No2.Ant = prox
582
583             prox = aresta.No2.Prox
584             aresta.No2.Prox = aresta.No1
585             aresta.No1.Ant = prox
586
587         else:
588             aresta.No1.Prox = aresta.No2
589             aresta.No2.Ant = aresta.No1
590
591             aresta.No2.Prox = aresta.No1
592             aresta.No1.Ant = aresta.No2
593
594     def _criarNoAresta(self, u):
595         """
596             Cria um novo nó para a lista de adjacência do vértice u.
597         """
598         return self.NoAresta(u)
599
600     def _criarAresta(self, u, v):
601         """
602             Cria uma nova aresta entre os vértices u e v.
603         """
604         aresta = self.Aresta()
605         aresta.v1 = u
606         aresta.v2 = v
607         aresta.No1 = self.NoAresta(u)
608         aresta.No2 = self.NoAresta(v)
609
610         return aresta
611
612     def _removerAresta(self, u, v):
613         """
614             Remove a aresta entre os vértices u e v.
615             Se a aresta não existir, não há efeitos colaterais.
616         """
617         aresta = self._acharAresta(u, v)
618
619         if aresta is None:
620             return
621
622         self._removerVizinho(u, v, aresta)
623         self._removerVizinho(v, u, aresta)
624
625         self._atualizarProximos(aresta)
626
627     def _acharAresta(self, u, v):
628         """
629             Busca a aresta entre os vértices u e v.
630             Se a aresta não existir, retorna None.
631         """
632         aresta = None
633
634         for i in range(1, self.n + 1):
635             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
636                 aresta = self.L[i].e
637                 break
638
639         return aresta
640
641     def _adicionarVizinho(self, u, v, aresta):
642         """
643             Adiciona o vértice v como vizinho do vértice u.
644             A aresta é adicionada à lista de vizinhos do vértice u.
645         """
646         self.L[u].Viz = v
647         self.L[u].e = aresta
648
649         aresta.No1 = self.NoAresta(u)
650         aresta.No2 = self.NoAresta(v)
651
652     def _removerVizinho(self, u, v, aresta):
653         """
654             Remove o vértice v como vizinho do vértice u.
655             A aresta é removida da lista de vizinhos do vértice u.
656         """
657         self.L[u].Viz = None
658         self.L[u].e = None
659
660         aresta.No1 = None
661         aresta.No2 = None
662
663     def _atualizarProximos(self, aresta):
664         """
665             Atualiza os próximos apontamentos para a aresta.
666             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
667         """
668         if self.VizinhancaDuplamenteLigada:
669             prox = aresta.No1.Prox
670             aresta.No1.Prox = aresta.No2
671             aresta.No2.Ant = prox
672
673             prox = aresta.No2.Prox
674             aresta.No2.Prox = aresta.No1
675             aresta.No1.Ant = prox
676
677         else:
678             aresta.No1.Prox = aresta.No2
679             aresta.No2.Ant = aresta.No1
680
681             aresta.No2.Prox = aresta.No1
682             aresta.No1.Ant = aresta.No2
683
684     def _criarNoAresta(self, u):
685         """
686             Cria um novo nó para a lista de adjacência do vértice u.
687         """
688         return self.NoAresta(u)
689
690     def _criarAresta(self, u, v):
691         """
692             Cria uma nova aresta entre os vértices u e v.
693         """
694         aresta = self.Aresta()
695         aresta.v1 = u
696         aresta.v2 = v
697         aresta.No1 = self.NoAresta(u)
698         aresta.No2 = self.NoAresta(v)
699
700         return aresta
701
702     def _removerAresta(self, u, v):
703         """
704             Remove a aresta entre os vértices u e v.
705             Se a aresta não existir, não há efeitos colaterais.
706         """
707         aresta = self._acharAresta(u, v)
708
709         if aresta is None:
710             return
711
712         self._removerVizinho(u, v, aresta)
713         self._removerVizinho(v, u, aresta)
714
715         self._atualizarProximos(aresta)
716
717     def _acharAresta(self, u, v):
718         """
719             Busca a aresta entre os vértices u e v.
720             Se a aresta não existir, retorna None.
721         """
722         aresta = None
723
724         for i in range(1, self.n + 1):
725             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
726                 aresta = self.L[i].e
727                 break
728
729         return aresta
730
731     def _adicionarVizinho(self, u, v, aresta):
732         """
733             Adiciona o vértice v como vizinho do vértice u.
734             A aresta é adicionada à lista de vizinhos do vértice u.
735         """
736         self.L[u].Viz = v
737         self.L[u].e = aresta
738
739         aresta.No1 = self.NoAresta(u)
740         aresta.No2 = self.NoAresta(v)
741
742     def _removerVizinho(self, u, v, aresta):
743         """
744             Remove o vértice v como vizinho do vértice u.
745             A aresta é removida da lista de vizinhos do vértice u.
746         """
747         self.L[u].Viz = None
748         self.L[u].e = None
749
750         aresta.No1 = None
751         aresta.No2 = None
752
753     def _atualizarProximos(self, aresta):
754         """
755             Atualiza os próximos apontamentos para a aresta.
756             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
757         """
758         if self.VizinhancaDuplamenteLigada:
759             prox = aresta.No1.Prox
760             aresta.No1.Prox = aresta.No2
761             aresta.No2.Ant = prox
762
763             prox = aresta.No2.Prox
764             aresta.No2.Prox = aresta.No1
765             aresta.No1.Ant = prox
766
767         else:
768             aresta.No1.Prox = aresta.No2
769             aresta.No2.Ant = aresta.No1
770
771             aresta.No2.Prox = aresta.No1
772             aresta.No1.Ant = aresta.No2
773
774     def _criarNoAresta(self, u):
775         """
776             Cria um novo nó para a lista de adjacência do vértice u.
777         """
778         return self.NoAresta(u)
779
780     def _criarAresta(self, u, v):
781         """
782             Cria uma nova aresta entre os vértices u e v.
783         """
784         aresta = self.Aresta()
785         aresta.v1 = u
786         aresta.v2 = v
787         aresta.No1 = self.NoAresta(u)
788         aresta.No2 = self.NoAresta(v)
789
790         return aresta
791
792     def _removerAresta(self, u, v):
793         """
794             Remove a aresta entre os vértices u e v.
795             Se a aresta não existir, não há efeitos colaterais.
796         """
797         aresta = self._acharAresta(u, v)
798
799         if aresta is None:
800             return
801
802         self._removerVizinho(u, v, aresta)
803         self._removerVizinho(v, u, aresta)
804
805         self._atualizarProximos(aresta)
806
807     def _acharAresta(self, u, v):
808         """
809             Busca a aresta entre os vértices u e v.
810             Se a aresta não existir, retorna None.
811         """
812         aresta = None
813
814         for i in range(1, self.n + 1):
815             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
816                 aresta = self.L[i].e
817                 break
818
819         return aresta
820
821     def _adicionarVizinho(self, u, v, aresta):
822         """
823             Adiciona o vértice v como vizinho do vértice u.
824             A aresta é adicionada à lista de vizinhos do vértice u.
825         """
826         self.L[u].Viz = v
827         self.L[u].e = aresta
828
829         aresta.No1 = self.NoAresta(u)
830         aresta.No2 = self.NoAresta(v)
831
832     def _removerVizinho(self, u, v, aresta):
833         """
834             Remove o vértice v como vizinho do vértice u.
835             A aresta é removida da lista de vizinhos do vértice u.
836         """
837         self.L[u].Viz = None
838         self.L[u].e = None
839
840         aresta.No1 = None
841         aresta.No2 = None
842
843     def _atualizarProximos(self, aresta):
844         """
845             Atualiza os próximos apontamentos para a aresta.
846             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
847         """
848         if self.VizinhancaDuplamenteLigada:
849             prox = aresta.No1.Prox
850             aresta.No1.Prox = aresta.No2
851             aresta.No2.Ant = prox
852
853             prox = aresta.No2.Prox
854             aresta.No2.Prox = aresta.No1
855             aresta.No1.Ant = prox
856
857         else:
858             aresta.No1.Prox = aresta.No2
859             aresta.No2.Ant = aresta.No1
860
861             aresta.No2.Prox = aresta.No1
862             aresta.No1.Ant = aresta.No2
863
864     def _criarNoAresta(self, u):
865         """
866             Cria um novo nó para a lista de adjacência do vértice u.
867         """
868         return self.NoAresta(u)
869
870     def _criarAresta(self, u, v):
871         """
872             Cria uma nova aresta entre os vértices u e v.
873         """
874         aresta = self.Aresta()
875         aresta.v1 = u
876         aresta.v2 = v
877         aresta.No1 = self.NoAresta(u)
878         aresta.No2 = self.NoAresta(v)
879
880         return aresta
881
882     def _removerAresta(self, u, v):
883         """
884             Remove a aresta entre os vértices u e v.
885             Se a aresta não existir, não há efeitos colaterais.
886         """
887         aresta = self._acharAresta(u, v)
888
889         if aresta is None:
890             return
891
892         self._removerVizinho(u, v, aresta)
893         self._removerVizinho(v, u, aresta)
894
895         self._atualizarProximos(aresta)
896
897     def _acharAresta(self, u, v):
898         """
899             Busca a aresta entre os vértices u e v.
900             Se a aresta não existir, retorna None.
901         """
902         aresta = None
903
904         for i in range(1, self.n + 1):
905             if self.L[i].Viz == v and self.L[i].e == self.NoAresta(v):
906                 aresta = self.L[i].e
907                 break
908
909         return aresta
910
911     def _adicionarVizinho(self, u, v, aresta):
912         """
913             Adiciona o vértice v como vizinho do vértice u.
914             A aresta é adicionada à lista de vizinhos do vértice u.
915         """
916         self.L[u].Viz = v
917         self.L[u].e = aresta
918
919         aresta.No1 = self.NoAresta(u)
920         aresta.No2 = self.NoAresta(v)
921
922     def _removerVizinho(self, u, v, aresta):
923         """
924             Remove o vértice v como vizinho do vértice u.
925             A aresta é removida da lista de vizinhos do vértice u.
926         """
927         self.L[u].Viz = None
928         self.L[u].e = None
929
930         aresta.No1 = None
931         aresta.No2 = None
932
933     def _atualizarProximos(self, aresta):
934         """
935             Atualiza os próximos apontamentos para a aresta.
936             Se a lista é duplamente ligada, os próximos apontamentos são alterados.
937         """
938         if self.VizinhancaDuplamenteLigada:
939             prox = aresta.No1.Prox
940             aresta.No1.Prox = aresta.No2
941             aresta.No2.Ant = prox
942
943             prox = aresta.No2.Prox
944             aresta.No2.Prox = aresta.No1
945             aresta.No1.Ant = prox
946
947         else:
948             aresta.No1.Prox = aresta.No2
949             aresta.No2.Ant = aresta.No1
950
951             aresta.No2.Prox = aresta.No1
952             aresta.No1.Ant = aresta.No2
953
954     def _criarNoAresta(self, u):
955         """
956             Cria um novo nó para a lista de adjacência do vértice u.
957         """
958         return self.NoAresta(u)
959
960     def _criarAresta(self, u, v):
961         """
962             Cria uma nova aresta entre os vértices u e v.
963         """
964         aresta = self.Aresta()
965         aresta.v1 = u
966         aresta.v2 = v
967         aresta.No1 = self.NoAresta(u)
968         aresta.No2 = self.NoAresta(v)
969
970         return aresta
971
972     def _removerAresta(self, u, v):
973         """
974             Remove a aresta entre os vértices u e v.
97
```

```

41     """
42     Adiciona aresta uv.
43     """
44     def AdicionarLista(u,v,e,Tipo):
45         No = GrafoListaAdj.NoAresta()
46         No.Viz, No.e, No.Prox, self.L[u].Prox = v, e, self.L[u].Prox, No
47         if self.VizinhancaDuplamenteLigada:
48             self.L[u].Prox.Ant = self.L[u]
49             if self.L[u].Prox.Prox != None:
50                 self.L[u].Prox.Prox.Ant = self.L[u].Prox
51         if self.orientado:
52             No.Tipo = Tipo
53         return No
54
55     e = GrafoListaAdj.Aresta()
56     e.v1, e.v2 = u, v
57     e.No1 = AdicionarLista(u,v,e,"+")
58     e.No2 = AdicionarLista(v,u,e,"-")
59     self.m = self.m+1
60     return e
61
62
63     def RemoverAresta(self, uv):
64         """
65         Remove a aresta uv.
66         """
67         def RemoverLista(No):
68             No.Ant.Prox = No.Prox
69             if No.Prox != None:
70                 No.Prox.Ant = No.Ant
71             RemoverLista(uv.No1)
72             RemoverLista(uv.No2)
73
74     def SaoAdj(self, u, v):
75         """
76         Retorna True se uv é uma aresta e False, caso contrário.
77         """
78         Tipo = "+" if self.orientado else "*"
79         for w in self.N(u, Tipo):
80             if w == v:
81                 return True
82         return False
83
84     def N(self, v, Tipo = "*", Fechada=False, IterarSobreNo=False):
85         """
86         Retorna lista de Grafo.NoAresta representando os vizinhos do vértice v.
87         Se Fechada=True, o próprio v é incluído na lista.
88         Tipo="*" significa listar todas as arestas incidentes em v. Se G é orientado,
89         Tipo)+" (resp. "-") significa listar apenas as arestas de saída (resp. de entrada) de ↓
90         v.
91         IterarSobreNo=False indica que a lista de vizinhos deve constituir da lista de ↓
92         vértices. Caso
93         contrário, a lista é dos nós da lista encadeada de vizinhos (NoAresta).
94         """
95         if Fechada:
96             No = GrafoListaAdj.NoAresta()

```

```

95     No.Viz, No.e, No.Prox = v, None, None
96     yield No if IterarSobreNo else No.Viz
97     w = self.L[v].Prox
98     while w != None:
99         if Tipo == "*" or w.Tipo == Tipo:
100            yield w if IterarSobreNo else w.Viz
101            w = w.Prox

```

### 2.10.1 Algoritmo 2.1: Centro de Árvore

O programa a seguir corresponde à implementação do Algoritmo 2.1. As Linhas 6–9 computam o vetor  $d$  de modo que  $d[w]$  corresponda ao grau do vértice  $w$ . Tal vetor servirá para obter a lista  $F$  das folhas da árvore  $T$ , que serão em seguida removidas. Durante a remoção, se o vértice vizinho da folha sendo removida se torna por sua vez uma folha, ele é inserido na lista  $Flin$ . Ao final da remoção de todas as folhas de  $T$ ,  $Flin$  consiste na lista de folhas da árvore resultante. Sendo assim, toma-se  $Flin$  por  $F$  e repete-se o processo, até que o centro seja encontrado.

Programa 2.1: Determinação do centro de uma árvore

```

#Algoritmo 2.1: Determinação do centro de uma árvore
#Dados: árvore T
def CentroArvore(T):
    if T.n == 1:
        return [1]
    d = [0]*(T.n+1)
    n = T.n
    for (u,v) in T.E():
        d[u]=d[u]+1;d[v]=d[v]+1
    F = [ v for v in T.V() if d[v]==1 ]

    while n > 2:
        Flin = []
        for f in F:
            v_no = next(T.N(f,IterarSobreNo=True)) #por ser folha, tem apenas um vizinho
            v = v_no.Viz
            d[v]=d[v]-1; n=n-1; T.RemoverAresta(v_no.e)
            if d[v] == 1:
                Flin.append(v)
        F = Flin

    return F

```

## 2.11 Exercícios

2.1 Construir uma representação geométrica do grafo

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

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

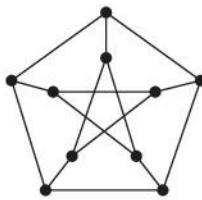


Figura 2.40: O grafo de Petersen

2.2 Em todo grafo  $G$ , dois caminhos de comprimento máximo possuem, pelo menos, um vértice comum. Provar ou dar contra-exemplo, para os seguintes casos:

- (i)  $G$  é desconexo.
- (ii)  $G$  é conexo.

2.3 Caracterizar os grafos  $G(V, E)$ , para os quais centro  $(G) = V$ .

2.4 Caracterizar os grafos  $G(V, E)$  nos seguintes casos:

- (i) centro( $G$ ) = centro( $G - s$ ), para todo  $s \in V - \text{centro}(G)$ .
- (ii) centro( $G$ ) = centro( $G - S$ ), para todo  $S \subseteq V - \text{centro}(G)$ .

2.5 Seja  $G(V, E)$  um grafo cujo comprimento do maior caminho simples é  $k$ . Então um vértice  $v \in V$  pertence ao centro( $G$ ) se e somente se excentricidade  $(v) = \lfloor k/2 \rfloor$ . Provar ou dar contra-exemplo.

2.6 Mostrar que todo grafo conexo com um número mínimo de arestas é uma árvore.

2.7 Caracterizar as árvores cujo centro possui exatamente um vértice.

2.8 Dados um grafo  $G(V, E)$  e um subconjunto de arestas  $E' \subseteq E$  determinar as condições para que exista uma árvore geradora de  $G$ , que não contenha arestas de  $E$ .

2.9 O grafo *bloco-articulação*  $B(G)$  de um dado grafo  $G$  é o grafo bipartido, cujos vértices são os blocos  $B_i$  e as articulações  $a_j$  de  $G$ , respectivamente. Um par de vértices  $(B_j, a_j)$  é adjacente quando a articulação  $a_j$  for parte do bloco  $B_i$ . Mostrar que (i)  $B(G)$  é uma árvore, e (ii) a distância entre duas folhas de  $B(G)$  é par.

2.10 Construir o grafo bloco-articulação do grafo da Figura 2.21(a).

2.11 Toda folha do grafo bloco-articulação  $B(G)$  corresponde a algum bloco de  $G$ , o qual contém algum vértice que não é articulação de  $G$ . Provar ou dar contra-exemplo.

2.12 Mostrar que o *grafo de Petersen*, ilustrado na Figura 2.40, não é planar.

- 2.13 Um grafo  $G(V, E)$  é *maximal planar* quando for planar, e para todo par de vértices não adjacentes  $v, w \in V$ , o grafo  $G + (v, w)$  é não planar. Mostrar que toda face de um grafo maximal planar é um triângulo.
- 2.14 Um grafo planar é *outerplanar* quando todos os seus vértices pertencerem a uma mesma face. Todo grafo outerplanar é hamiltoniano. Provar ou dar contra-exemplo.
- 2.15 Provar que cada um dos grafos ilustrados nas Figuras 2.20(a), 2.26(d) e 2.27 é não hamiltoniano.
- 2.16 Provar que todo grafo bipartido com número ímpar de vértices não é hamiltoniano.
- 2.17 Construir um grafo que seja 3-conexo, planar e não hamiltoniano.
- 2.18 Mostrar através de um exemplo que a condição do Teorema 2.9 não é suficiente.
- 2.19 Mostrar através de um exemplo que a condição do Teorema 2.10 não é necessária.
- 2.20 Construir um grafo que possua conectividade de vértices 2, conectividade de arestas 3, número cromático 3 e que seja regular de grau 3.
- 2.21 Construir um grafo planar, regular de grau 3, biconexo e não hamiltoniano.
- 2.22 Construir o menor grafo sem triângulos, cujo número cromático seja 3.
- 2.23 Construir um grafo sem triângulos, cujo número cromático seja 4.
- 2.24 Construir um grafo sem triângulos, cujo número cromático seja igual a um inteiro dado  $k$ .
- 2.25 Um grafo  $G(V, E)$  é *maximal  $k$ -colorível* quando  $G$  for  $k$ -colorível e para todo par de vértices distintos  $v, w \in V$ , o grafo  $G + (v, w)$  não é  $k$ -colorível. Mostrar que  $G$  é maximal 2-colorível se e somente se  $G$  for bipartido completo.
- 2.26 Provar que todo grafo  $G$  admite subgrafo  $\chi(G)$ -crítico.
- 2.27 Provar o Lema 2.8.
- 2.28 Caracterizar os digrafos acíclicos  $D$  para os quais o fecho e a redução transitivos de  $D$  são isomorfos.
- 2.29 Um grafo  $G$  é de *comparabilidade* quando  $G$  for o grafo subjacente de um digrafo acíclico  $D$ , fechado transitivamente. Provar que os vértices que formam o maior caminho de  $D$  induzem a maior clique de  $G$ .

- 2.30 Um *torneio* é um digrafo cujo grafo subjacente é completo (e sem arestas paralelas). Todo torneio não acíclico é hamiltoniano. Provar ou dar contra-exemplo.
- 2.31 Todo torneio possui um caminho hamiltoniano. Provar ou dar contra-exemplo.
- 2.32 Seja  $R$  uma matriz de adjacências de um grafo  $G$ . Provar que o elemento  $(i, j)$  da matriz  $R^k$  fornece o número de caminhos distintos de comprimento  $k$ , em  $G$ , do vértice  $v_i$  ao  $v_j$ .
- 2.33 Seja  $G(V, E)$  um grafo. Defina a matriz  $S = (s_{ij})$ ,  $|E| \times |E|$ , como:  
 $s_{ij} = 1 \Leftrightarrow$  as arestas  $e_i, e_j$  são adjacentes,  
 $s_{ij} = 0$ , caso contrário.  
Então  $S$  é uma representação de  $G$ . Provar ou dar exemplo.
- 2.34 Seja  $G(V, E)$  um grafo não direcionado, representado por uma matriz de adjacências. Descrever um algoritmo para realizar as seguintes tarefas:
- Dados dois vértices  $v_i, v_j \in V$ , determinar se eles são adjacentes ou não.
  - Dado o vértice  $v_i \in V$ , encontrar o conjunto de vizinhos de  $v_i$ .
- Determinar a complexidade para a realização de cada tarefa (i) e (ii), separadamente.
- 2.35 Repetir o Exercício 2.34, supondo que a representação seja a de matriz de incidências, ao invés de matriz de adjacências.
- 2.36 Repetir o Exercício 2.34, supondo que a representação seja a de estrutura de adjacências, ao invés de matriz de adjacências.
- 2.37 POSCOMP 2012
- Seja  $G = (V, E)$  um grafo em que  $V$  é o conjunto de vértices e  $E$  é o conjunto de arestas. Com base nesse grafo, considere as afirmativas a seguir.
- Se  $G$  é o  $K_{3,3}$  então o número cromático de  $G$  é 3.
  - Se  $G$  é o  $K_{3,3}$  então, retirando-se uma aresta de  $G$ , o grafo se torna planar.
  - Se  $G$  é o  $K_{2,2}$  então  $G$  é um grafo euleriano e hamiltoniano ao mesmo tempo.
  - Se  $G$  é um  $K_{n,n}$  então  $G$  tem um conjunto independente máximo igual a  $n$ .
- Assinale a alternativa correta.
- Somente as afirmativas I e II são corretas.
  - Somente as afirmativas I e IV são corretas.
  - Somente as afirmativas III e IV são corretas.
  - Somente as afirmativas I, II e III são corretas.
  - Somente as afirmativas II, III e IV são corretas.

### 2.38 POSCOMP-2014

Considerando que um grafo possui  $n$  vértices e  $m$  arestas, assinale a alternativa que apresenta, corretamente, um grafo planar.

- a)  $n = 5, m = 10$
- b)  $n = 6, m = 15$
- c)  $n = 7, m = 21$
- d)  $n = 8, m = 12$
- e)  $n = 9, m = 22$

### 2.39 UVA Online Judge 117

Faça um algoritmo para o seguinte problema:

O mapa de uma região de uma cidade é um grafo onde os vértices são as esquinas e as arestas os trechos de rua entre as esquinas. Um carteiro atende determinada região e tem que andar em todos os trechos de rua da região. No grafo que corresponde à região do carteiro a maioria dos vértices tem grau par, mas dois deles podem ter grau ímpar. É dado esse grafo, bem como o custo de andar por cada trecho de rua. Determinar o custo mínimo para o carteiro percorrer todos os trechos de rua pelo menos uma vez.

## 2.12 Notas Bibliográficas

A literatura sobre grafos é relativamente vasta. Há diversos textos de caráter geral. Entre outros, Berge (1962 e 1973), Bollobás (1979). Bondy e Murty (1976), Carré (1979), Deo (1974), Harary (1969), Ore (1962), Wilson (1972). Entre os textos mais recentes, destacamos Bondy e Murty (2008), Diestel (1997), Bollobás (1998), West (1995). Os seguintes são livros sobre temas mais específicos. Entre outros, Tutte (1966), conectividade. Ore (1967) e Saaty e Kainen (1977), problema das quatro cores. Bollobás (1978), e Fritsch e Fritsch (1998), grafos extremos. Busacker e Saaty (1965), e Harary, Norman e Cartwright (1965), grafos direcionados. Golumbic (2004), grafos perfeitos. Capobianco e Molluzzo (1978) apresentam exemplos e contra-exemplos para diversos teoremas e conjecturas. Christofides (1975), Even (1979) e Gondran e Minoux (1979) tratam de grafos sob enfoque algorítmico. Wilson e Beineke (1979) é uma coletânea de aplicações de grafos em uma diversidade de outras áreas. Lovász (1979) apresenta problemas e exercícios, com sugestões e soluções. Brandstadt, Le e Spinrad (1999), classes de grafos. Spinrad (2003), representações em grafos. McKee e McMorris (1999), grafos de interseção. Golumbic e Trenk (2004), grafos de tolerância. Diestel (1990), decomposição em grafos. Lovász e Plummer (2009), emparelhamentos. Entre os livros editados em Portugal, mencionamos o texto de Cardoso, Szymanski e Rostami (2009), o qual cobre uma parte considerável da teoria, e Simões Pereira (2009). A literatura brasileira em grafos inclui os textos de Andrade (1980), Barbosa (1974), Boaventura Netto (2001), Furtado (1973), Lucchesi (1979), Savulecu (1980). Um texto sobre igualdade min-max em grafos, em língua portuguesa é o de Feofiloff e Lucchesi (1988). Como mencionado, os trabalhos pioneiros em árvores foram de Kirchhoff (1847), Cayley (1857) e Jordan (1869), devendo-se incluir, também, os resultados sobre enumeração de árvores de Cayley (1889). A importância das árvores para algoritmos é ressaltada em Knuth (1997). O Teorema 2.6 é de Dirac (1960), enquanto que o Teorema 2.7 é de Whitney (1932). Esta caracterização constitui uma variação do teorema de Menger (1927). Diversas outras variações desse teorema foram desenvolvidas. O Teorema 2.9, fundamental em planaridade, é de Kuratowski (1930). Representações planas de grafos, tais que todas suas linhas sejam retas, foram construídas por Fáry (1948). O Teorema 2.11 é de Dirac (1952). Um texto sobre grafos hamiltonianos em língua portuguesa é de Wakabayashi (1977). Grafos  $k$ -críticos foram inicialmente estudados por Dirac (1952a). As provas dos Teoremas 2.9 a 2.11 foram escritas a partir de Bondy e Murty (2008). Conforme também já mencionado, o teorema das quatro cores foi provado por Appel e Haken (1977) e (1977a). Os grafos bloco-articulação, Exercícios 2.9 a 2.11, foram caracterizados por Gallai (1964), e Harary e Prins (1966). O grafo do Exercício 2.12 foi introduzido em Petersen (1891). Os Exercícios 2.23 e 2.24 são da construção de Mycielski (1955), o qual obtém grafos desprovidos de triângulos e com um dado

número cromático arbitrário. Grafos de comparabilidade, Exercício 2.29, foram inicialmente caracterizados em Ghouilla-Houri (1962). Gilmore e Hoffman (1964) e Gallai (1967). O Exercício 2.31 corresponde ao primeiro resultado para a classe dos digrafos torneios, descrito em Redei (1934).

# TÉCNICAS BÁSICAS

---

## 3.1 Introdução

Neste capítulo examinam-se as primeiras técnicas que podem ser aplicadas para o desenvolvimento de algoritmos em grafos. Na maior parte dos casos, essas técnicas encontram-se presentes em algoritmos de forma combinada com outros processos. Sua importância maior reside na grande frequência em que, em geral, são empregadas.

## 3.2 Processo de Representação

Em geral, um algoritmo para resolver um certo problema em um grafo supõe que este esteja representado sob uma forma adequada. Por outro lado, seria também conveniente que o grafo fosse fornecido ao algoritmo sob uma maneira simples de ser especificado. É razoável, por exemplo, que essa especificação corresponda aos seus conjuntos de vértices e arestas, respectivamente. Um problema básico portanto consiste em construir a representação desejada a partir desses dois conjuntos. Nesta seção apresenta-se um algoritmo para construir uma estrutura de adjacências de um digrafo, a partir de seus conjuntos de vértices e arestas. Esta representação foi selecionada por sua larga aplicação em implementações de algoritmos.

Seja  $D(V, E)$  um digrafo dado. Denote por  $p$  o vetor que fornece os ponteiros iniciais para as listas de adjacência de  $D$ . Isto é,  $p(v)$  indica o primeiro elemento (vértice), se houver, da lista  $A(v)$ . Essa é naturalmente composta pelos vértices  $u_i$  adjacentes a  $v$ . Associado a cada vértice  $u_i \in A(v)$  existe um ponteiro  $t_i$  que informa a localização do próximo vértice na lista  $A(v)$ . A inexistência de próximo elemento em  $A(v)$  é denotada por  $\emptyset$ . A estrutura de dados necessária consiste pois em um vetor  $p$ , o qual é formado por  $n$  elementos e de vetores  $u$  e  $t$ , com  $m$  elementos cada, conforme mencionado na Seção 2.9. Sem perda de generalidade, supõe-se que o conjunto de vértices do digrafo dado seja  $V = \{1, 2, \dots, n\}$ , sendo conhecido de antemão o seu número de arestas  $m$ .

---

**Algoritmo 3.1:** Estrutura de adjacências de um digrafo

**Dados:**  $n, m, E$

```
para  $i = 1, \dots, n$  efetuar
     $p(i) := \emptyset$ 
    para  $i = 1, \dots, m$  efetuar
        seja  $(v, w)$  a  $i$ -ésima aresta de  $E$ 
         $u(i) := w$ 
         $t(i) := p(v)$ 
         $p(v) := i$ 
```

---

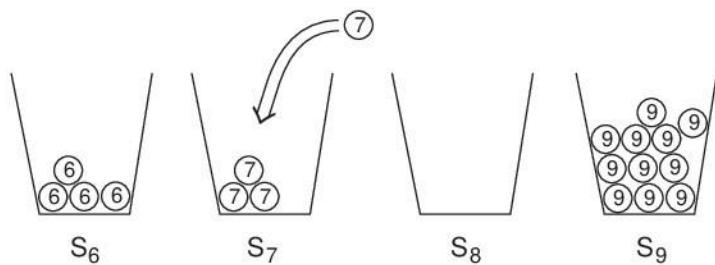


Figura 3.1: Ordenação por caixas

Ao final do processo acima as listas de adjacências do digrafo  $D$  estarão construídas. A complexidade desse algoritmo é obviamente  $O(n + m)$ . Deste ponto em diante do texto, convencionar-se que todo algoritmo descrito contém como passo inicial um algoritmo para a construção da representação desejada. Isto é, se nos *dados* do algoritmo descrito for especificado um grafo  $G(V, E)$ , por exemplo, significa  $G(V, E)$  na representação conveniente.

### 3.3 Adjacências

A técnica mais elementar que se pode aplicar para se examinar um grafo consiste na análise das listas de adjacências de seus vértices. Na realidade, esta técnica é tanto elementar quanto poderosa, pois várias outras técnicas mais sofisticadas e empregadas em grafos podem ser decompostas em operações mais elementares, as quais correspondem ao exame de listas de adjacências de vértices.

Na técnica de adjacência, em geral existe uma propriedade  $P(v)$  definida de modo apropriado, para vértices  $v \in V$  de um grafo  $G(V, E)$ . A ideia consiste em se examinar listas de adjacências  $A(v)$ , para  $v \in V$ , de modo que se possa distinguir se  $w \in A(v)$  satisfaz ou não à propriedade  $P(v)$ . Por exemplo,  $P(v)$  poderia ser uma propriedade do tipo: “existe algum  $w \in A(v)$  que pertença a um certo conjunto  $C(v)$ ?” O exemplo apresentado na Seção 3.5 é dessa natureza.

### 3.4 Ordenação de Vértices ou Arestas

Esta técnica consiste na ordenação de vértices ou arestas de  $G$ , segundo um certo critério. O critério depende, naturalmente, da aplicação em questão. São bastante comuns, por exemplo, ordenações de vértices segundo os seus respectivos graus. Ou então ordenações das listas de adjacências segundo os graus dos vértices que as compõem, respectivamente. Arestas podem ser ordenadas de acordo com pesos a elas atribuídos, ou então lexicograficamente, segundo rótulos dos vértices correspondentes que as formam. E assim por diante.

Uma observação importante é que nos algoritmos em grafos, frequentemente é vantajoso utilizar um processo muito simples de ordenação, denominado *ordenação por caixas*. Seja  $S$  um conjunto de números inteiros a ser ordenado. Sejam  $a$  e  $b$  os elementos mínimo e máximo de  $S$ , respectivamente. Defina os subconjuntos, inicialmente vazios,  $S_a, S_{a+1}, \dots, S_b$ , denominados *caixas*. O algoritmo de ordenação é simples. Cada elemento de  $S$  de valor  $j$  é inserido na caixa  $S_j$ . Note-se que existe precisamente uma única caixa, onde cada elemento deve ser inserido. Após o processo de inserção ter sido completado para todos os elementos, estes são retirados das respectivas caixas, na ordem  $S_a, S_{a+1}, \dots, S_b$  (Figura 3.1). A identificação da caixa onde cada elemento de  $S$  deve ser inserido pode ser realizada em tempo constante. A retirada de cada elemento da respectiva caixa também é uma operação de tempo constante. Essas etapas requerem, portanto,  $O(n)$  passos, para todo o processo. Por outro lado, cada uma das  $b - a + 1$  caixas é manipulada na ordenação. Logo, a complexidade do algoritmo é  $O(n + b - a)$ .

No caso de ordenação de vértices, segundo os seus graus correspondentes, por exemplo, o valor máximo que  $|b - a|$  pode alcançar é  $n - 1$ . Assim sendo, a complexidade dessa ordenação é linear com o número de vértices (observe, porém, que para calcular os graus de todos os vértices são necessárias  $O(n + m)$  operações).

### 3.5 Coloração Aproximada

O problema da determinação exata do número cromático de um grafo  $G$  é, em geral, bastante difícil. Muitas vezes, porém, o interesse é obter apenas alguma aproximação para o problema. Nesse caso, o objetivo consiste em procurar alguma coloração de vértices “razoável” para o grafo, isto é, cujo número de cores esteja, se possível, próximo do número cromático de  $G$ . Nessa seção, apresenta-se um algoritmo aproximativo para o problema de coloração em grafos. Ele ilustra uma aplicação para as técnicas descritas nas duas seções anteriores.

O seguinte algoritmo simples pode ser utilizado. No passo inicial, dado  $G(V, E)$ , seleciona-se um vértice qualquer  $v_1 \in V$  e atribui-se a  $v_1$  a cor 1. No passo geral, os vértices  $v_1, \dots, v_{j-1}$  já foram examinados e coloridos, sendo  $k \geq 1$  o total de cores utilizadas. Seja  $C_i$  o conjunto de vértices de cor  $i$ . Considere agora o vértice  $v_j$ . Se existir alguma cor  $i$  tal que  $C_i \cap A(v_j) = \emptyset$  então ao vértice  $v_j$  pode ser atribuída a cor  $i$ . Caso contrário, atribuir a cor  $k + 1$  a  $v_j$ .

Examinando o algoritmo anterior, observa-se que se  $v_j$  for um vértice de grau alto em relação aos demais, a chance de ocorrer  $C_i \cap A(v_j) \neq \emptyset$ , seria intuitivamente maior. Portanto, na tentativa de melhorar a aproximação obtida pelo processo, talvez fosse razoável considerar os vértices em ordem não crescente de grau. Essa observação conduz ao seguinte algoritmo.

---

#### Algoritmo 3.2: Coloração aproximada

---

**Dados:** grafo  $G(V, E)$

ordenar  $V$  em ordem não crescente  $v_1, \dots, v_n$  de graus

$C_1 := C_2 := \dots := C_n := \emptyset$

colorir  $v_1$  com a cor 1 (incluir  $v_1$  em  $C_1$ )

**para**  $j = 2, \dots, n$  **efetuar**

$r := \min\{i | A(v_j) \cap C_i = \emptyset\}$

    colorir  $v_j$  com a cor  $r$  (incluir  $v_j$  em  $C_r$ ).

---

#### Lema 3.1

O Algoritmo 3.2 está correto.

**Prova** Para cada vértice  $v_j$  de cor  $i$ , assegurou-se  $A(v_j) \cap C_i = \emptyset$ . ■

Uma implementação direta desse algoritmo requer  $O(n^2)$  operações, pois a determinação de cada  $r$ , a partir da expressão indicada no algoritmo ( $r := \min\{i | A(v_j) \cap C_i = \emptyset\}$ ) consome  $O(n)$  passos. Contudo, é possível uma formulação mais eficiente, como se segue. Define-se um vetor  $f$ , com  $f(k) = j$  indicando que o vértice  $v_j$  é adjacente a algum outro vértice de cor  $k$ . Para colorir  $v_j$ , a ideia consiste em consultar o vetor  $f$ , restringindo-se o exame aos seus  $|A(v_j)|$  elementos iniciais, no máximo. Essas observações conduzem ao Algoritmo 3.3. A coloração de um vértice  $v_j$  com a cor  $r$  é indicada por  $\text{cor}(v_j) = r$ .

O número de operações efetuadas para colorir cada vértice  $v_j$ , no bloco definido pelo laço “*para  $j = 1, \dots, n$  efetuar*” é  $O(|A(v_j)|)$ , pois necessariamente  $f(r) \neq j$  quando  $r > |A(v_j)|$ . A ordenação de  $V$  em ordem não crescente de graus pode ser efetuada em tempo  $O(n)$ , através do método de ordenação por caixas. A determinação do grau de cada vértice  $v_j$  requer  $O(|A(v_j)|)$  operações. Logo a complexidade do Algoritmo 3.3 é  $O(n + \sum |A(v_j)|) = O(n + m)$ .

Como exemplo, a aplicação do algoritmo ao grafo da Figura 3.2(a) produz a coloração indicada, que utiliza 3 cores. Esta coloração não é ótima, pois há uma outra com 2 cores somente, como na Figura 3.2(b).

O objetivo inicial de formular um algoritmo que produzisse colorações com um total de cores próximo ao número cromático não foi alcançado, pois é possível formular exemplos em que as colorações obtidas são “arbitrariamente ruins”. E, talvez surpreendentemente, todos os algoritmos aproximativos conhecidos para o problema de coloração possuem em comum esta baixa qualidade.

**Algoritmo 3.3:** Coloração aproximada

**Dados:** grafo  $G(V, E)$

ordenar  $V$  em ordem não crescente  $v_1, \dots, v_n$  de graus

**para**  $j = 1, \dots, n$  efetuar

$$f(j) := 0$$

**para**  $j = 1, \dots, n$  efetuar

**para**  $w \in A(v_j)$  **efetuar**

**se** *w* já colorido **então**

$$f(\text{cor}(w)) := j$$

= 1

enqua

se  $f(r) \neq j$  então

$$\text{cor}(v_i) := r$$

so contrário

*case contrario*

卷之三

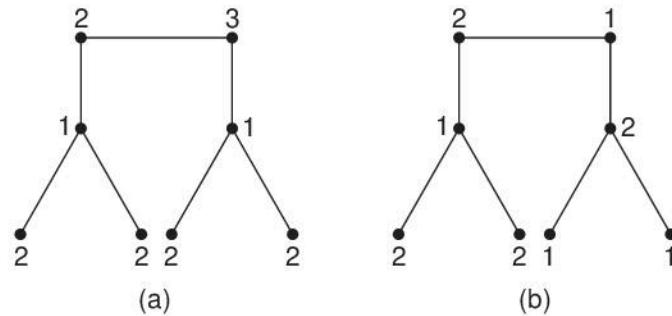


Figura 3.2: Colorações aproximada (a) e ótima (b)

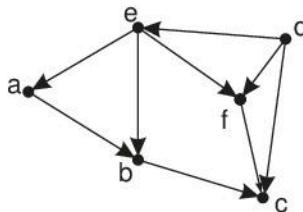


Figura 3.3: Exemplo para o Algoritmo 3.3

### 3.6 Ordenação Topológica

Conforme mencionado na Seção 2.8, todo digrafo acíclico  $D(V, E)$  induz um conjunto parcialmente ordenado  $(V, <)$ , definido por:

$$v < w \Leftrightarrow v \text{ alcança } w \text{ em } D, \text{ para todo } v, w \in V, v \neq w.$$

Baseando-se nessa relação, torna-se simples mostrar que é possível ordenar os vértices do digrafo de modo a obter uma sequência  $v_1, \dots, v_n$  satisfazendo:

$$v_i < v_j \Rightarrow i < j, \text{ para } 1 \leq i, j \leq n.$$

Tal sequência é denominada *ordenação topológica*. Ela se constitui na maneira natural de dispor os vértices de um digrafo acíclico em uma linha reta, de modo que todas as suas arestas estejam direcionadas da esquerda para a direita. Vários algoritmos envolvendo digrafos acíclicos possuem como passo intermediário a computação de uma ordenação topológica.

O algoritmo descrito a seguir admite como entrada o digrafo  $D(V, E)$  e produz como saída uma ordenação topológica  $v_1, \dots, v_n$ , dos vértices de  $D$ . A ação consiste em excluir do digrafo e dar saída a todo vértice  $w$  que possua grau de entrada nulo. Cada exclusão produz um novo digrafo. A ação é repetida para esse novo digrafo e assim por diante, até que não hajam mais vértices a considerar. A seguinte formulação implementa a ideia.

---

#### Algoritmo 3.4: Ordenação topológica

**Dados:** digrafo acíclico  $D(V, E)$

**para**  $j = 1, \dots, n$  **efetuar**

escolher um vértice  $w$  com grau de entrada nulo em  $D$

retirar de  $D$  o vértice  $w$  e as arestas dele divergentes

definir  $v_j := w$

---

#### Lema 3.2

O algoritmo 3.4 está correto.

**Prova** *Indução em  $j$ .* Observe que como  $D$  é acíclico, existe necessariamente pelo menos um vértice de  $D$  com grau de entrada nulo. ■

De um modo geral, a ordenação topológica não é única. No algoritmo este fato se reflete na liberdade existente para a escolha do vértice  $w$ , dentre aqueles que possuem grau de entrada nulo.

Como exemplo, o Algoritmo 3.4 quando aplicado ao digrafo da Figura 3.3, produz a ordenação topológica  $d, e, f, a, b, c$ , supondo que na 3<sup>a</sup> iteração fosse realizada a escolha  $w = f$ .

Utilizando uma variável que indica o valor do grau de entrada de cada vértice ao longo do processo, é possível implementar o algoritmo em tempo  $O(n + m)$ . Combinando-se as técnicas de ordenação, da Seção 3.4 e a da presente, pode-se ordenar topologicamente todas as listas de adjacências de um digrafo, em um tempo total  $O(n + m)$ . Isto corresponde a dispor as listas de adjacências de forma tal que se  $w_1, w_2 \in A(v)$  e  $w_1 < w_2$  então  $w_1$  precede  $w_2$  em  $A(v)$ , para todo vértice  $v$  do digrafo.

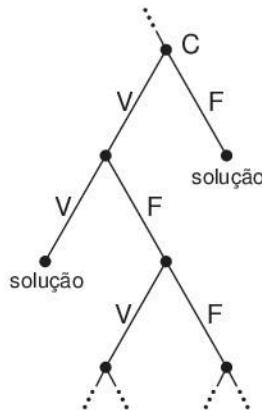


Figura 3.4: Árvore de decisão

### 3.7 Recursão

A técnica de recursão é largamente empregada em algoritmos. Basicamente consiste na decomposição de um problema dado em subproblemas menores. Esses últimos, por sua vez, são também compostos em subproblemas ainda menores e assim por diante, até que se tornem tão pequenos que seja trivial resolvê-los. Uma vez resolvidos todos os subproblemas, a solução do problema é obtida através de uma composição das soluções dos subproblemas correspondentes. Naturalmente, as formas de decomposição e composição dependem de cada caso. Contudo, e sempre que possível, deve-se procurar decompor o problema em subproblemas de tamanhos idênticos e tão pequenos quanto se possa.

Um exemplo clássico é o cálculo do factorial de  $n$ , utilizando a seguinte recursão:

$\text{fatorial}(n) :=$   
se  $n = 0$  então 1  
caso contrário  $n \cdot \text{fatorial}(n - 1)$ .

A recursão é conhecida também por outros nomes, dependendo da área em que é utilizada. Em matemática, corresponde à *recorrência*. Na computação, muitas vezes recebe o nome de *dividir-para-conquistar*. Exemplos de recursão aparecem, neste texto, combinados com a utilização de outras técnicas.

### 3.8 Árvores de Decisão

O conhecimento de limites inferiores para um problema algorítmico é, naturalmente, um dado essencial para que se possa avaliar a dificuldade de resolver o problema. Nessa seção, descreve-se uma técnica que permite o seu cálculo, em alguns casos. Para ilustrar a técnica, apresenta-se, na seção seguinte, um exemplo de como determinar um limite inferior do problema de ordenação.

Seja  $P$  um problema e  $\alpha$  um algoritmo que resolve  $P$ , de tal modo que a operação dominante em  $\alpha$  seja a comparação. Isto é, o número de comparações que  $\alpha$  executa no processo exprime a sua complexidade. Cada comparação é de natureza binária, ou seja, admite exatamente duas alternativas como resposta. A técnica seguinte se destina a determinar um limite inferior para as complexidades que  $\alpha$  pode apresentar.

O efeito das comparações em uma computação de  $\alpha$  pode ser modelado através de uma árvore estritamente binária  $T$ , denominada *árvore de decisão*. Cada comparação  $C$  efetuada no processo é univocamente representada por um vértice interior  $v$  de  $T$ . Os filhos esquerdo e direito correspondem as duas alternativas (*verdadeiro* ou *falso*) do resultado de  $C$ . Este depende da entrada de  $\alpha$ , naturalmente. Um resultado verdadeiro de  $C$ , por exemplo, conduz a uma nova comparação  $C'$ , representada em  $T$  pelo filho esquerdo de  $v$ . Os dois filhos de  $C'$  correspondem por sua vez, respectivamente, às alternativas do resultado de  $C'$ , e assim por diante (Figura 3.4). A árvore  $T$  contém todas as alternativas do processo, para todas as entradas possíveis de  $\alpha$ .

As folhas de  $T$  correspondem ao conjunto dos estados finais a que a computação de  $\alpha$  pode conduzir. Ou seja, a cada possível saída diferente de  $\alpha$  deve corresponder pelo menos uma folha diferente de  $T$ . Assim sendo, o comprimento do caminho em  $T$  desde a raiz até uma folha é igual ao número de comparações efetuadas por  $\alpha$ , para uma certa entrada  $E$ . Este número exprime a complexidade local assintótica de  $\alpha$ , relativa a  $E$ . Consequentemente, a altura de  $T$  é igual ao número de comparações que  $\alpha$  efetua no pior caso. Ou seja, é igual à complexidade de  $\alpha$ . Como decorrência, um limite inferior para a altura de  $T$  é também um limite inferior para  $P$ . Seja  $h$  a altura mínima dentre as árvores de decisão correspondente a todos os algoritmos  $\alpha$  que resolvem  $P$ . O limite inferior máximo de  $P$  é igual a  $h$ .

### 3.9 Limite Inferior para Ordenação

Como aplicação da modelagem das comparações de um algoritmo, através de uma árvore de decisão, mostra-se a seguir que  $\Omega(n \log n)$  é um limite inferior para o problema de ordenação de uma sequência  $S$  com  $n$  elementos. Esse resultado é restrito a algoritmos de ordenação em que as comparações sejam operações dominantes. A existência de um tal algoritmo com complexidade  $O(n \log n)$  assegura que esse limite inferior é máximo. Seja  $\alpha$  um algoritmo de ordenação nas condições acima e cuja entrada é  $S$ . Seja  $T$  a árvore de decisão de altura  $h$ , correspondente a  $\alpha$ . O número de folhas de  $T$  é  $\geq n!$ , pois a saída de  $\alpha$  pode corresponder a qualquer permutação de sua entrada. Assim sendo,  $T$  possui no máximo  $2^h$  folhas, pois  $T$  é uma árvore binária. Logo,

$$2^h \geq n! \Rightarrow h \geq \log(n!). \text{ Como } n > 0, \\ n! = n(n-1) \cdots 1 \geq n(n-1) \cdots \lceil n/2 \rceil > (n/2)^{n/2}.$$

Consequentemente,  $h > (n/2)(\log n - 2)$ . Isto é,  $h$  é  $\Omega(n \log n)$ .

Ou seja,  $\Omega(n \log n)$  é um limite inferior para a altura de  $T$ . De acordo com a seção anterior,  $\Omega(n \log n)$  é também um limite inferior para o problema de ordenação, considerando algoritmos baseados em comparações. Por outro lado, são conhecidos algoritmos de ordenação com complexidade  $O(n \log n)$ . Logo esses algoritmos são ótimos, e, portanto, o limite inferior  $\Omega(n \log n)$  é máximo.

### 3.10 Programas em Python

Esta seção contém as implementações de algoritmos formulados neste capítulo. O programa segue as descrições gerais apresentadas nos Capítulos 1 e 2.

#### 3.10.1 Algoritmo 3.3: Coloração Aproximada

O Programa 3.1 corresponde à implementação do Algoritmo 3.3. As Linhas 4–6 computam o vetor  $d$  de modo que  $d[w]$  corresponda ao grau do vértice  $w$ . Em seguida, cria-se uma lista  $V$  de pares ordenados  $(d(v), v)$  para todo  $v \in V(G)$ , onde  $d(v)$  representa o grau de  $v$ . Esta lista é então ordenada descendentemente na Linha 8. Note que, em Python, uma tupla é menor do que outra se a primeira é lexicograficamente menor que a segunda. Isto significa que os vértices dos pares ordenados serão em particular ordenados pelos respectivos graus de forma não crescente. Na sequência, a lista  $V$  é redefinida para conter apenas os vértices dos pares ordenados, mantendo a ordem após a ordenação, que servirá como sequência de vértices a considerar no processo de coloração. O restante da implementação é tradução direta do algoritmo.

Programa 3.1: Coloração aproximada

```

1 #Algoritmo 3.3: Coloração aproximada
2 #Dados: grafo G
3 def ColoracaoAproximada(G):
4     d = [0] * (G.n+1)
5     for (u,v) in G.E():
6         d[u]=d[u]+1;d[v]=d[v]+1

```

```

7   V = [(d[v],v) for v in G.V()]
8   V.sort(reverse=True)
9   V = [v[1] for v in V]
10  f = [0]*(G.n+1)
11  cor = [None]*(G.n+1)
12  chi = 0
13  for j in V:
14      for w in G.N(j):
15          if cor[w] != None:
16              f[cor[w]] = j
17          r = 1
18          while cor[j]==None:
19              if f[r] != j:
20                  cor[j] = r; chi = max(chi,r)
21              else:
22                  r = r + 1
23
24  return chi

```

### 3.10.2 Algoritmo 3.4: Ordenação Topológica

O Programa 3.2 corresponde à implementação do Algoritmo 3.4. A entrada do algoritmo é o digrafo D. As Linhas 6–8 computam o vetor d de modo que  $d[w]$  corresponda ao grau de entrada do vértice  $w$ . Durante tal computação, as Linhas 9–10 se encarregam de inserir na pilha Q todos os vértices com grau de entrada nulo. Cada iteração da repetição da Linha 11 determina o vértice  $v_j$  da ordenação topológica, para todo  $1 \leq j \leq n$ . Com efeito, a Linha 12 remove um vértice com grau de entrada nulo da pilha Q, que será definido como  $v_j$  (e armazenado em  $V[j-1]$ ). Em seguida, a remoção de  $v_j$  implica a diminuição em uma unidade no grau de entrada de cada vértice na vizinhança de saída de  $v_j$  (Linhas 14–15). Quando tal grau de entrada se torna zero, o respectivo vértice passa a poder ser escolhido como próximo elemento da ordenação topológica e, portanto, entra na pilha Q (Linhas 16–17).

Programa 3.2: Ordenação topológica

```

1 #Algoritmo 3.4: Ordenação topológica
2 #Dado: digrafo D
3 def OrdenacaoTopologica(D):
4     d = [0]*(D.n+1)
5     Q = []; V = [None]*D.n
6     for w in D.V():
7         for v in D.N(w, "-"):
8             d[w] = d[w]+1
9             if d[w] == 0:
10                 Q.append(w)
11     for j in range(1, D.n+1):
12         w = Q.pop()
13         V[j-1] = w
14         for v in D.N(w, "+"):
15             d[v] = d[v]-1
16             if d[v] == 0:
17                 Q.append(v)
18     return V

```

## 3.11 Exercícios

- 3.1 Modificar o Algoritmo 3.1, de modo a construir uma estrutura de adjacências para um grafo não direcionado.

- 3.2 Seja  $G(V, E)$  um grafo não direcionado, representado por uma matriz de adjacências,  $V' \subseteq V$  um subconjunto de vértices e  $E' \subseteq E$  um subconjunto de arestas de  $G$ . Descrever um algoritmo para realizar as seguintes tarefas:
- (i) Construir o subgrafo induzido em  $G$ , pelo subconjunto de vértices  $V'$ , na mesma representação considerada, isto é, matriz de adjacências.
  - (ii) Construir o subgrafo induzido em  $G$ , pelo subconjunto de arestas  $E'$ , na mesma representação considerada, isto é, matriz de adjacências.
  - (iii) Determinar a complexidade para a realização de cada tarefa (i) e (ii), separadamente.
- 3.3 Repetir o Exercício 3.2, supondo que a representação seja a de matriz de incidências, ao invés de matriz de adjacências.
- 3.4 Repetir o Exercício 3.2, supondo que a representação seja a de estrutura de adjacências, ao invés de matriz de adjacências.
- 3.5 Descrever um algoritmo de ordenação por caixas, para realizar uma ordenação alfabética. Se existirem  $n$  ítems a serem ordenados, cada qual com  $m$  caracteres alfabéticos, qual será a complexidade do processo?
- 3.6 Apresentar um exemplo de um grafo  $G$ , com número cromático  $\chi$  e tal que o Algoritmo 3.2 de coloração aproximada atribua aos vértices de  $G$  um total de  $2\chi$  ou mais cores.
- 3.7 Os algoritmos 3.2 e 3.3 produzem colorações rigorosamente iguais. Provar ou dar contra-exemplo.
- 3.8 Seja uma variação do Algoritmo 3.2, de coloração aproximada, na qual é omitida a operação de ordenação de vértices do grafo, segundo os seus graus. Apresentar exemplos de grafos, para os quais a aproximação obtida através desta variação é melhor do que aquela do Algoritmo 3.2.
- 3.9 Um digrafo admite ordenação topológica se e somente se for acíclico. Provar ou dar contra-exemplo.
- 3.10 Seja  $R$  uma matriz de adjacências de um digrafo acíclico  $D$ , construída segundo uma permutação de seus vértices que corresponde a uma ordenação topológica. Mostrar que  $R$  é uma matriz triangular.
- 3.11 Seja  $D(V, E)$  um digrafo tal que contenha exatamente um ciclo simples  $C$ . Seja  $V_1 \subseteq V$  o subconjunto dos vértices de  $D$  alcançáveis de algum vértice de  $C$ . Se  $D$  é a entrada do Algoritmo 3.4 de ordenação topológica, qual será a saída correspondente?
- 3.12 Seja  $T$  o número de ordenações topológicas distintas de um digrafo  $D(V, E)$ . Quais são os valores mínimo e máximo de  $T$ ?
- 3.13 Seja  $A$  uma árvore direcionada enraizada. Determine o número de ordenações topológicas distintas dos vértices de  $A$ .

- 3.14 Seja  $T$  a árvore de decisão correspondente a um algoritmo  $\alpha$ , baseado em ordenações, para resolver certo problema  $P$ . Então o nível mínimo de uma folha de  $T$  é igual ao número de comparações que  $\alpha$  efetua no melhor caso, provar ou dar contra-exemplo.

### 3.15 UVA Online Judge 103

Escreva um algoritmo para o seguinte problema:

Dadas  $n$  “caixas”, cada uma com  $m$  dimensões, deseja-se saber qual a maior pilha de “caixas” que pode ser formada,  $c_1, c_2, \dots, c_k$ , tal que cada caixa  $c_i$  pode ser encaixada na caixa  $c_{i+1}$  ( $1 \leq i < k$ ). Formalmente, definimos a propriedade de encaixamento como se segue. A caixa  $D = (d_1, d_2, \dots, d_m)$  encaixa-se na caixa  $E = (e_1, e_2, \dots, e_m)$  se existir uma permutação de  $1 \dots m$ , tal que  $d_{\pi(i)} < e_i$  ( $1 \leq i \leq m$ ).

### 3.16 UVA Online Judge 11686

Escreva um algoritmo para o seguinte problema:

O jogo de pega-varetas é fascinante. Um conjunto de  $n$  varetas coloridas, numeradas de 1 a  $n$ , é jogado em uma mesa resultando em um emaranhado de varetas, umas por cima das outras. Cada jogador, por turno, retira quantas varetas puder, só podendo retirar varetas livres. Dadas  $n$  varetas e a disposição das mesmas na mesa, na forma de  $m$  pares  $(a, b)$ , que indicam que a vareta  $a$  está sobre a vareta  $b$ , determinar uma ordem válida para a retirada das varetas. Se isso não for possível, indicar com uma mensagem.

### 3.17 UVA Online Judge 10305

Escreva um algoritmo para o seguinte problema:

João tem  $n$  tarefas para fazer, numeradas de 1 a  $n$ . Infelizmente, as tarefas não são independentes e pode ser que a execução de cada uma delas só seja possível após a realização de outras. Dadas  $n$  tarefas e as relações de precedência entre elas, indicar uma possível ordem para sua execução.

## 3.12 Notas Bibliográficas

Métodos de ordenação foram considerados, com certo detalhe, em diversos trabalhos. A obra clássica no assunto é Knuth (1973). O Algoritmo 3.3 é de Matula, Marble e Isaacson (1972). O efeito da ordenação dos vértices na qualidade de certas aproximações para o problema de coloração é discutido em Matula (1968). Estes algoritmos efetuam uma coloração baseados em uma ordenação dos vértices. Historicamente, mencionamos os algoritmos de Welsh e Powel (1967), e Brélaz (1979), para este tipo de técnica. Um outro algoritmo de coloração aproximada foi descrito posteriormente por Halldórsson (1993). A produção de exemplos desfavoráveis para algoritmos aproximativos de coloração é tratado em Mithchen (1976). A dificuldade existente para formular uma boa aproximação para este problema é apresentada em Garey e Johnson (1976). Um dos primeiros estudos de algoritmos de coloração, em língua portuguesa, é Santos (1979). O livro de Jensen e Toft (1995) contém diversos problemas de coloração de grafos. Um algoritmo de complexidade linear para o problema de ordenação topológica foi descrito inicialmente em Knuth (1997). Veja também Kahn (1962), Kase (1963) e Lasser (1961). A técnica da recursão é largamente empregada na formulação de algoritmos. Knuth (1997) é uma fonte de exemplos. Um texto específico no assunto é Barron (1968). A tradução de recursão para iteração é também discutida em Knuth (1974a). Árvores de decisão são ferramentas bastante utilizadas em mineração de dados, por exemplo. Métodos recursivos são frequentemente empregados na construção das árvores de decisão. Neste sentido, veja a referência Quinlan (1986).

---

# BUSCAS EM GRAFOS

---

## 4.1 Introdução

Dentre as técnicas existentes para a solução de problemas algorítmicos em grafos, a busca tem lugar de destaque, pelo grande número de problemas que podem ser resolvidos através da sua utilização. Provavelmente, a importância dessa técnica é ainda maior quando o universo das aplicações for restrito aos algoritmos considerados eficientes.

A busca visa resolver um problema básico, qual seja o de como explorar um grafo. Isto é, dado um grafo deseja-se obter um processo sistemático de como caminhar pelos vértices e arestas do mesmo. Por exemplo, quando o grafo é uma árvore, esta questão se torna mais simples. Pois uma das formas de nela caminhar pode ser definida, recursivamente, através da seguinte sequência de operações:

Se a árvore for vazia, não há nada a fazer. Caso contrário:

1. Visite a raiz da árvore.
2. Caminhe pela subárvore mais à esquerda da raiz,  
em seguida, pela 2<sup>a</sup> mais à esquerda,  
em seguida, pela 3<sup>a</sup> mais à esquerda,  
e assim por diante.

Este tipo de caminhamento é denominado *preordem* e sua aplicação à árvore da Figura 4.1 conduz à seguinte sequência de visitas:

$$a \ b \ d \ e \ i \ j \ c \ f \ k \ g \ h \ l \ n \ m$$

Observe que neste caminhamento se procura descer na árvore, tão profundo quanto possível, da esquerda para a direita, sistematicamente. Uma outra forma de se caminhar em árvores enraizadas é o processo denominado *ordem de nível*. Neste, o caminhamento é nível a nível, da esquerda para direita. A árvore da Figura 4.1 quando percorrida em ordem de nível produz a seguinte sequência de visitas:

$$a \ b \ c \ d \ e \ f \ g \ h \ i \ j \ k \ l \ m \ n$$

Naturalmente há outras formas de se caminhar em árvores.

Quando se transporta o mesmo problema para grafos, aflora de imediato uma dificuldade: não há um referencial geral a ser considerado. Em outras palavras, não são definidos, de forma absoluta, os conceitos de esquerda, direita e nível. Supondo, por exemplo, fixado o vértice  $c$  do grafo da Figura 4.2, como definir um caminhamento sistemático no grafo, de modo que fique determinado o próximo vértice a ser visitado na sequência de visitas?

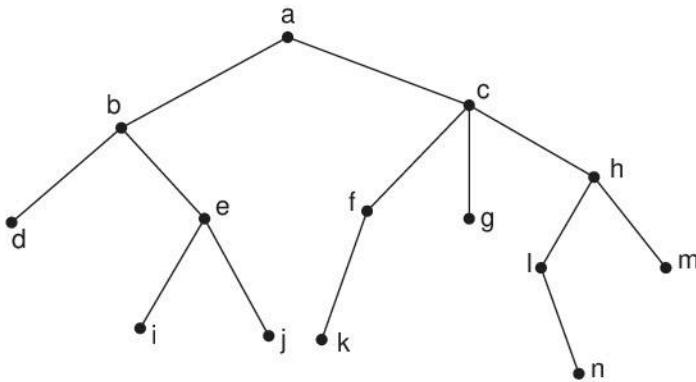


Figura 4.1: Uma árvore enraizada

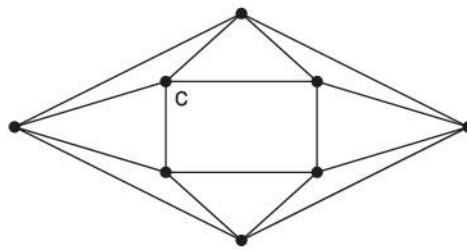


Figura 4.2: Como caminhar sistematicamente?

Em particular, como caminhar no grafo, de modo a visitar todos os vértices e arestas, evitando contudo repetições desnecessárias de visitas a um mesmo vértice ou aresta?

Esta última questão é de enorme dificuldade, de um modo geral, a não ser que sejam utilizados recursos adicionais, durante o caminhamento, que permitam reconhecer se um dado vértice ou aresta já foi ou não visitado anteriormente. Comumente, esses recursos adicionais correspondem a um conjunto de até  $n$  marcas. Uma marca é associada a um vértice  $v$  com a finalidade de registrar que  $v$  foi visitado. Isto permite distingui-la dos vértices não visitados. Na próxima seção o assunto é apresentado em detalhe.

## 4.2 Algoritmo Básico

Seja  $G$  um grafo conexo em que todos os seus vértices se encontram desmarcados. No passo inicial, marca-se um vértice arbitrariamente escolhido. No passo geral, seleciona-se algum vértice  $v$  que esteja marcado e seja incidente a alguma aresta  $(v, w)$  ainda não selecionada. A aresta  $(v, w)$  torna-se então selecionada e o vértice  $w$  marcado (caso ainda não o seja). O processo termina quando todas as arestas de  $G$  tiverem sido selecionadas. Esse tipo de caminhamento é denominado *busca* no grafo  $G$ .

Quando a aresta  $(v, w)$  é selecionada a partir do vértice marcado  $v$ , diz-se que  $(v, w)$  foi *explorada* e o vértice  $w$  *alcançado*. Um vértice torna-se *explorado* quando todas as arestas incidentes ao mesmo tiverem sido exploradas. Assim sendo, durante o processo de exploração de um vértice é possível que este venha a ser alcançado diversas vezes. Utiliza-se também o termo *visita* a arestas ou vértices, em lugar de exploração. O vértice inicial é denominado *raiz* da busca.

Como exemplo, considere efetuar uma busca no grafo da Figura 4.3. Escolhe-se um vértice inicial, por exemplo 1, e alguma aresta incidente ao mesmo, seja  $(1,2)$ . O vértice 1 torna-se marcado e a aresta  $(1,2)$  explorada. Nessa mesma ocasião, o vértice 2 torna-se também marcado. Seleciona-se, em seguida, algum vértice marcado, por exemplo 1, e alguma aresta incidente a 1 e ainda não explorada; por exemplo  $(1,4)$ , a qual se torna explorada. Isto produz a marcação do vértice 4. Seleciona-se, em seguida, por exemplo, o vértice 2 e a aresta  $(2,4)$ , tornando-a explorada. Observe que o vértice 4 já se encontrava marcado. Escolhe-se, em seguida, por exemplo, o próprio

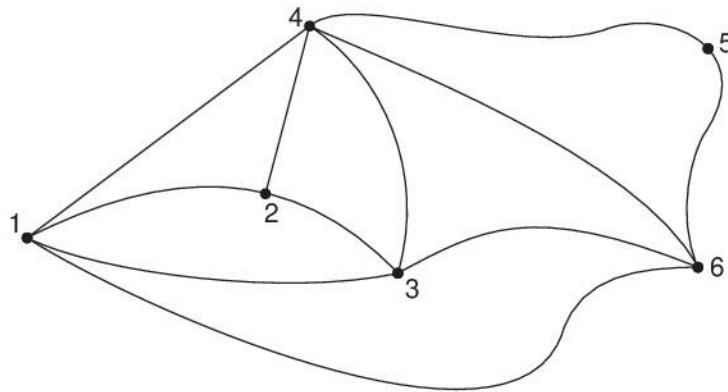


Figura 4.3: Exemplo para uma busca

vértice 4 e explora-se a aresta  $(4,6)$ , marcando-se o vértice 6. Escolhe-se agora o vértice 2, explora-se a aresta  $(2,3)$  e marca-se o vértice 3. Isto completa a exploração do vértice 2. Escolhe-se um novo vértice marcado e o processo continua até que tenha sido completada a exploração de todos os vértices (isto é, todas as arestas tenham sido exploradas).

Observe que um vértice  $v$  torna-se marcado precisamente quando a primeira aresta incidente a  $v$  é explorada. Nessa ocasião, é iniciada a exploração de  $v$ . Recorda-se que essa exploração é completada exatamente quando a última aresta incidente a  $v$  for por sua vez explorada.

Obviamente, a busca em um grafo não é única. Durante o processo há liberdade de escolha nas seguintes ocasiões.

No passo inicial:

*vértice inicial:* seleção do vértice inicial da busca.

No passo geral:

*vértice marcado:* seleção do vértice marcado  $v$ , a partir do qual se deseja explorar uma aresta  $(v, w)$  não explorada.

*aresta incidente:* seleção da aresta não explorada  $(v, w)$  incidente ao vértice marcado  $v$ .

---

#### Algoritmo 4.1: Busca geral

---

**Dados:** grafo  $G(V, E)$

desmarcar todos os vértices

escolher e marcar um vértice inicial

**enquanto** existir algum vértice  $v$  marcado e incidente a uma aresta  $(v, w)$  não explorada **efetuar**

  escolher o vértice  $v$  e explorar a aresta  $(v, w)$

  se  $w$  é não marcado **então**

    marcar  $w$

---

Numa busca geral essas escolhas são todas arbitrárias. Nas Seções 4.3 e 4.7, respectivamente, serão examinados critérios para a escolha de vértice marcado. Esses critérios correspondem à *busca em profundidade* e *busca em largura*, respectivamente. Em ambas, a escolha do próximo vértice marcado torna-se única. Contudo, para os casos de vértice inicial ou aresta incidente, não são conhecidos critérios gerais que conduzem a uma escolha sistemática, sem ambiguidades.

### 4.3 Busca em Profundidade

Uma busca é dita *em profundidade* quando o critério de escolha de vértice marcado (a partir do qual será realizada a próxima exploração de aresta) obedecer a:

“entre todos os vértices marcados e incidentes a alguma aresta ainda não explorada, escolher aquele *mais recentemente alcançado na busca*”.

Observe que a escolha de vértice marcado torna-se única e sem ambiguidade, segundo o critério apresentado. O seguinte algoritmo recursivo implementa esse processo.

---

**Algoritmo 4.2:** Busca em profundidade (utilizando pilha)

---

**Dados:**  $G(V, E)$ , conexo**Procedimento**  $P(v)$     marcar  $v$     colocar  $v$  na pilha  $Q$     **para**  $w \in A(v)$  **efetuar**        **se**  $w$  é não marcado **então**            visitar  $(v, w)$ 

▷ arestas visitadas (I)

 $P(w)$         **caso contrário**            **se**  $w \in Q$  e  $v, w$  não são consecutivos em  $Q$  **então**                visitar  $(v, w)$ 

▷ arestas visitadas (II)

            retirar  $v$  de  $Q$ 

desmarcar todos os vértices

    definir uma pilha  $Q$     escolher uma raiz  $s$      $P(s)$ 

Observe que no Algoritmo 4.2 são arbitrárias as escolhas da raiz da busca, bem como da aresta  $(v, w)$  a ser explorada, a partir do vértice marcado  $v$ . A escolha dessa aresta é dada implicitamente pela ordenação de  $A(v)$ , a qual é arbitrária.

O algoritmo em questão é recursivo, mas ao mesmo tempo utiliza uma pilha  $Q$ . Na realidade, isto representa, de certa forma, uma duplicação, pois a pilha faz o efeito da recursão. Assim é possível simplificar o algoritmo de modo a torná-lo puramente recursivo, sem o uso explícito da pilha  $Q$ , ou então não recursivo, somente com a utilização da pilha. A versão puramente recursiva é apresentada a seguir como Algoritmo 4.3. Nesta versão há necessidade de introduzir um parâmetro adicional  $u$ , além de um parâmetro adicional  $v$ , nas chamadas recursivas para evitar que uma aresta seja visitada mais de duas vezes.

O Algoritmo 4.3 divide o conjunto  $E$  das arestas de  $G$  em duas partes disjuntas: as arestas visitadas em (I) denominadas *arestas de árvore* e aquelas em (II), chamadas *arestas de retorno ou frondes*. Seja  $E_T$  o conjunto das arestas de árvore.

**Teorema 4.1**

O grafo  $T(V, E_T)$  é uma árvore geradora de  $G(V, E)$ .

**Prova** Como  $G$  é conexo, todo vértice  $v \in V$  é alcançado na busca. Logo,  $T$  é conexo. Seja uma aresta  $(v, w)$  e suponha  $v$  alcançado antes de  $w$ . Como  $(v, w) \in E_T$  se e somente se  $w$  estava desmarcado quando foi alcançado, conclui-se que a adição de  $(v, w)$  a  $E_T$  se dá simultaneamente com a adição de  $w$  a  $T$ . Logo,  $T$  não contém ciclos.

■

A árvore geradora  $(V, E_T)$  recebe o nome de *árvore de profundidade* de  $G$ . Normalmente, esta árvore será considerada como árvore enraizada, sendo seu vértice raiz, precisamente, a raiz  $s$  da busca.

**Algoritmo 4.3:** Busca em profundidade (recursivo)**Dados:**  $G(V, E)$ , conexo**Procedimento**  $P(v, u)$ marcar  $v$ **para**  $w \in A(v)$  **efetuar**    **se**  $w$  é não marcado **então**        visitar aresta de árvore  $(v, w)$ 

▷ arestas visitadas (I)

 $P(w, v)$     **caso contrário**        **se**  $w \neq u$  **então**            visitar aresta de retorno  $(v, w)$ 

▷ arestas visitadas (II)

desmarcar todos os vértices

        escolher uma raiz  $s$          $P(s, \emptyset)$ **Teorema 4.2**

Seja  $G(V, E)$  um grafo conexo e  $T$  uma árvore de profundidade de  $G$ , obtida a partir de uma busca em profundidade. Então toda aresta  $(v, w) \in E$  é tal que  $v$  é ancestral (ou descendente) de  $w$  em  $T$ .

**Prova** Sem perda de generalidade, suponha  $v$  alcançado antes de  $w$  em  $T$ . Então se  $v$  não é ancestral de  $w$  em  $T$ ,  $w$  só foi alcançado na busca após se retirar  $v$  da pilha  $Q$ . Mas isto contradiz o fato de que  $w$  é necessariamente alcançado quando  $v$  se torna o topo de  $Q$ , pois  $w \in A(v)$ . ■

Esse teorema pode ser generalizado como se segue:

**Teorema 4.3**

Seja  $G(V, E)$  um grafo conexo e  $T(V, E_T)$  uma árvore de profundidade de  $G$ . Então todo caminho  $C$  de  $G$  contém um vértice  $p$  tal que todos os vértices de  $C$  são descendentes de  $p$  em  $T$ .

**Prova** Seja  $C$  um caminho em  $G$  e  $p$  o vértice de  $C$  mais próximo à raiz  $s$ , em  $T$ . Suponha que  $C$  contém um vértice  $v$  qual não é descendente de  $p$ . Então necessariamente  $C$  contém uma aresta  $(v, w)$  tal que  $w$  é descendente de  $p$ , mas  $v$  não o é. Obviamente,  $v$  não é descendente de  $w$ , senão  $v$  o seria de  $p$ . Se, por outro lado,  $w$  é descendente de  $v$  então  $p$  e  $v$  estão no caminho de  $s$  a  $w$  em  $T$ . Nesse caso, a única possibilidade de acordo com as hipóteses consideradas é que  $p$  seja descendente de  $v$ , o que contradiz o fato de que  $p$  é o vértice de  $C$  mais próximo a  $s$ . Assim nenhum dentre  $v, w$  é descendente um do outro. Então  $T$  não é uma árvore de profundidade, uma contradição. ■

Observe que no algoritmo de busca em profundidade, cada aresta  $(v, w)$  é examinada exatamente duas vezes. Uma vez considerando  $w \in A(v)$  e a outra  $v \in A(w)$ . Este fato conduz à conclusão de que o processo de busca em profundidade possui complexidade  $O(n + m)$ .

Como exemplo, as Figuras 4.4(b) e 4.4(c) mostram buscas em profundidade diferentes, realizadas no grafo da Figura 4.4(a). As arestas desenhadas por linhas retas correspondem às arestas de árvore, enquanto que as curvas representam as frondes. A raiz em ambas as buscas é o vértice  $v_1$ .

A ordem em que os vértices são inseridos e/ou retirados da pilha  $Q$  é importante para diversas aplicações. Para cada  $v \in V$  e para uma dada busca em profundidade, define-se *profundidade de entrada de  $v$* ,  $PE(v)$ , e *profundidade de saída de  $v$* ,  $PS(v)$ , respectivamente, como sendo o número de ordem em que  $v$  foi inserido e retirado da pilha  $Q$ . Obviamente, as profundidades de entrada fornecem a sequência em que os vértices são alcançados pela primeira vez na busca. Se  $s$  é a raiz da busca, então *profundidade de entrada* ( $s$ ) = 1 e *profundidade de saída* ( $s$ ) =  $n$ . Supondo que  $v_3$  antecede  $v_5$  em  $A(v_4)$ , na busca do exemplo da Figura 4.4(b), seriam as seguintes as profundidades dos vértices desse grafo.

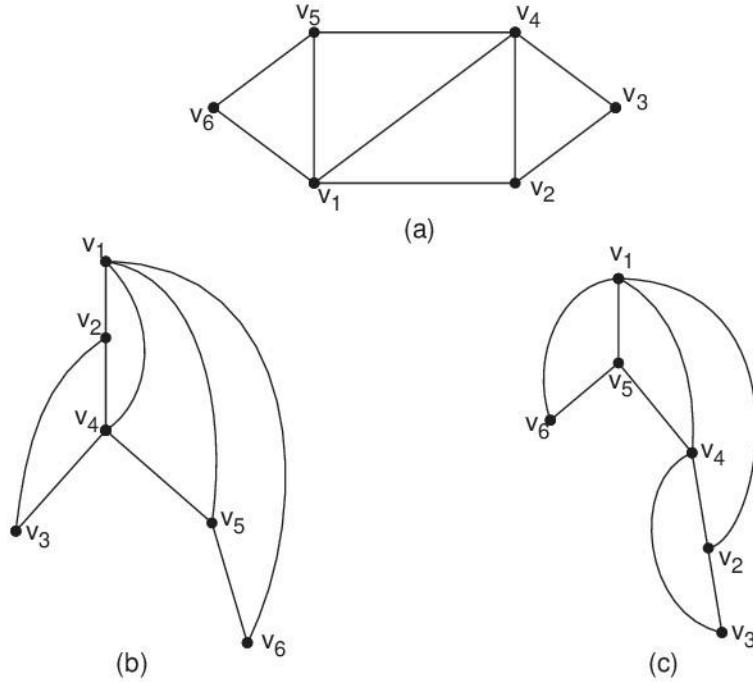


Figura 4.4: Busca em profundidade

vértice $v$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$
$\text{PE}(v)$	1	2	4	3	5	6
$\text{PS}(v)$	6	5	1	4	3	2

Finalmente, observa-se que os resultados apresentados nesta seção correspondem à busca em profundidade de grafos conexos. Caso o grafo considerado não seja conexo, os resultados se aplicam às suas componentes conexas, separadamente.

#### 4.4 Biconectividade

Como aplicação de busca em profundidade, será descrito um algoritmo para a determinação das componentes biconexas de um grafo. O processo determina também o conjunto dos vértices de articulação do grafo, como passo intermediário.

Seja  $G(V, E)$  um grafo e  $T(V, E_T)$  uma árvore de profundidade de  $G$ . Define-se a função  $\text{lowpt}:V \rightarrow V$ , do seguinte modo. Para cada vértice  $v \in V$ ,  $\text{lowpt}(v)$  é igual ao vértice mais próximo da raiz de  $T$  que pode ser alcançado a partir de  $v$ , caminhando-se em  $T$  para baixo, através de zero ou mais arestas de árvore e, em seguida, para cima utilizando no máximo uma aresta de retorno.

Como exemplo, a função  $\text{lowpt}$  correspondente ao grafo da Figura 4.5(b) é a seguinte:

$v$	1	2	3	4	5	6	7	8	9	10
$\text{lowpt}(v)$	8	8	2	2	2	8	8	8	8	1

A aplicação da função  $\text{lowpt}$  na determinação das articulações de um grafo é dada pelo Lema 4.1.

##### Lema 4.1

Seja  $G(V, E)$  um grafo conexo e  $T$  uma árvore de profundidade de  $G$ . Um vértice  $v \in V$  é uma articulação de  $G$  se e somente se

- (i)  $v$  é a raiz de  $T$  e  $v$  possui mais de um filho, ou

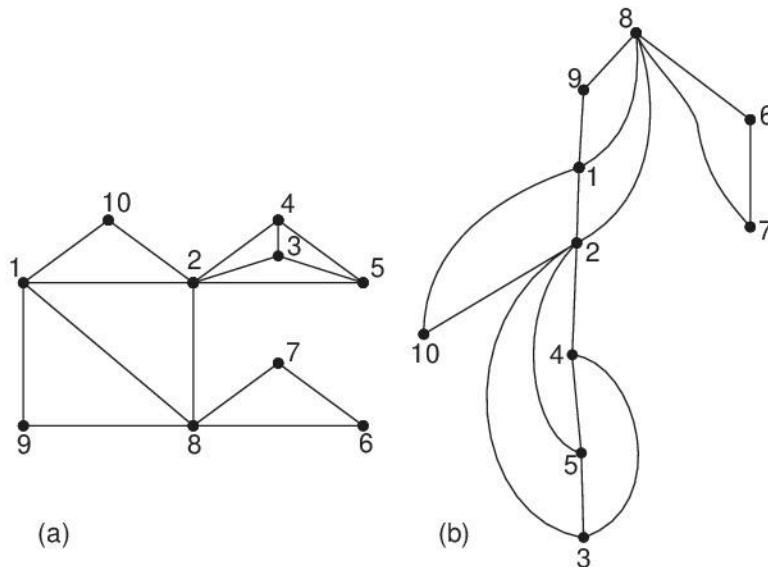


Figura 4.5: Outra busca em profundidade

(ii)  $v$  não é a raiz de  $T$ , e  $v$  possui um filho  $w$  tal que  $\text{lowpt}(w) = v$  ou  $w$ .

**Prova** Seja  $v$  uma articulação de  $G$ . Então existem vértices  $t, z \in V$  tais que todo caminho entre  $t$  e  $z$  contém  $v$ . Seja  $T$  uma árvore de profundidade de  $G$ . Seja  $u$  o ancestral comum a  $t$  e  $z$  mais afastado da raiz de  $T$ . (i) Suponha que  $v$  é a raiz de  $T$ . Então se  $u \neq v$  existe um caminho entre  $t$  e  $z$  que não contém  $v$ , uma contradição. Logo  $u$  e  $v$  coincidem, o que implica que  $t$  e  $z$  pertencem a subárvore diferentes de raiz  $v$ . Ou seja,  $v$  deve possuir dois filhos, no mínimo. (ii) Suponha que  $v$  não seja a raiz de  $T$ . Porque  $v$  é articulação,  $v$  possui pelo menos um filho  $w$  tal que não há aresta (fronde) entre um descendente de  $w$  e um ascendente próprio de  $v$ . Logo,  $v$  possui um filho  $w$  tal que  $\text{lowpt}(w) = v$  ou  $w$ . Para provar a suficiência, suponha que (i) é válido. Então todo caminho entre dois filhos de  $v$  contém  $v$ , ou seja,  $v$  é articulação. Suponha que (ii) é válido. Então todo caminho de  $w$  à raiz  $T$  contém  $v$ . Logo  $v$  é articulação. ■

Pelo Lema 4.1, conclui-se que é possível determinar as articulações de  $G$ , através dos valores  $\text{lowpt}(v)$ , para cada  $v \in V$ . Para se calcular os valores de  $\text{lowpt}$ , define-se para cada vértice  $v \in V$ , a função  $g(v)$  como sendo o ascendente  $w$  de  $v$  mais alto em  $T$ , tal que  $(v, w)$  é uma aresta de retorno. Se não houver algum ascendente nessas condições então  $g(v) = v$ , por definição. Os valores de  $\text{lowpt}(v)$  são então computados do seguinte modo:

**se**  $v$  é uma folha **então**

$$\text{lowpt}(v) := g(v)$$

**caso contrário**

$$\text{lowpt}(v) := \text{vértice mais próximo à raiz dentre } g(v) \text{ e } \text{lowpt}(u), \text{ para todo filho } u \text{ de } v.$$

A determinação de  $\text{lowpt}$  pode também ser obtida por uma busca em profundidade em complexidade  $O(n+m)$ . Para tal, basta percorrer uma vez a árvore da busca, de baixo para cima. Isto é, um vértice da árvore somente deve ser considerado quando todos os seus filhos já o tiverem sido.

Para determinar as componentes biconexas a partir dos vértices articulação, novamente recorre-se à busca em profundidade.

Sejam  $v, w$  vértices de  $G$ ,  $v$  pai de  $w$  em  $T$  e  $\text{lowpt}(w) = v$  ou  $w$ . Então  $w$  é chamado *demarcador* de  $v$ . Uma articulação é pai de um ou mais demarcadores. Além disso, todos os filhos da raiz da busca são também demarcadores. Eles podem ser determinados sem dificuldade, dadas a árvore  $T$  e a função  $\text{lowpt}$ .

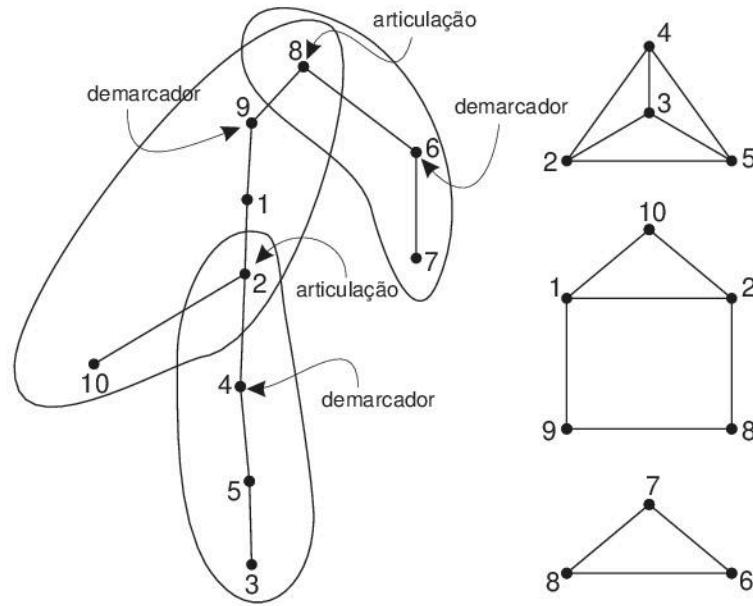


Figura 4.6: Determinação das componentes biconexas de um grafo

**Lema 4.2**

Seja  $G(V, E)$  um grafo e  $T$  uma árvore de profundidade de  $G$ . Sejam  $v, w \in V$ , com  $w$  um demarcador de  $v$  tal que a subárvore  $T_w$  de  $T$  com raiz  $w$  não contém articulações de  $G$ . Então os vértices de  $T_w$ , juntamente com  $v$ , induzem uma componente biconexa em  $G$ .

**Prova** O único vértice de  $G$  não pertencente à subárvore  $T_w$  e adjacente a algum vértice de  $T_w$  é precisamente a articulação  $v$ . ■

O algoritmo seguinte, para determinação das componentes biconexas de um dado grafo  $G$ , é uma aplicação sucessiva do Lema 4.2. No passo inicial, calculam-se as articulações e demarcadores de  $G$ , através de uma busca em profundidade que produz a árvore  $T$ . No passo geral, escolhe-se um demarcador  $w$  tal que a subárvore  $T_w$  de  $T$ , com raiz  $w$ , não possua articulações de  $G$ . O vértice  $v$  juntamente com os de  $T_w$  induzem uma componente biconexa em  $G$ . Retirar  $T_w$  de  $T$ . Além disso, se  $w$  é o único demarcador de  $v$  em  $T$ , doravante não considerar  $v$  como articulação. O processo se repete até que não haja mais demarcadores. Esse algoritmo pode ser implementado em tempo  $O(n + m)$ , percorrendo-se a árvore  $T$  de baixo para cima.

Como exemplo, considere novamente o grafo  $G$  da Figura 4.5(a), cujas articulações são os vértices 2 e 8, respectivamente. A Figura 4.6(a) ilustra uma árvore de profundidade  $T$  de  $G$ , com as articulações e demarcadores assinalados. A subárvore  $T_4$ , de raiz 4, não contém articulações de  $G$ . Logo,  $\{2, 4, 5, 3\}$  constituem os vértices de uma componente biconexa. Retirando-se  $\{4, 5, 3\}$  de  $T$  o vértice 2 deixa de ser articulação, pois 4 era seu único demarcador. O vértice 9 é agora um demarcador tal que  $T_9$  não contém articulações. Logo,  $\{8, 9, 1, 2, 10\}$  induzem uma componente biconexa em  $G$ . Retira-se  $\{9, 1, 2, 10\}$  de  $T$ . O vértice 6 é um demarcador e  $T_6$  não possui articulações. Entre os vértices de  $\{8, 6, 7\}$  induzem também uma componente biconexa em  $G$ . Retira-se  $\{6, 7\}$  de  $T$ . Todos os demarcadores de  $G$  já foram considerados, o que encerra o processo. A Figura 4.6(b) mostra as componentes obtidas.

## 4.5 Busca em Profundidade – Digrafos

A ideia básica da busca em profundidade em grafos direcionados é análoga ao caso não direcionado. A partir de um vértice  $s = v_1$  denominado *raiz*, constroem-se caminhos  $Q = v_1, \dots, v_k$ . Se existe algum vértice  $w$  divergente de  $v_k$  tal que  $w$  nunca pertenceu a algum caminho  $Q$ , então  $w$  é incluído em  $Q$ , o qual se torna  $v_1, \dots, v_k, w$  e o processo é repetido. Caso contrário, se não existe  $w$  nessas condições, o vértice  $v_k$  é retirado do caminho,

transformando-o em  $Q = v_1, \dots, v_{k-1}$ , e o processo é repetido. O término da busca se dá quando o vértice  $v_1$  é retirado de  $Q$ . Este processo é denominado *busca em profundidade de raiz*  $v_1$ .

Observa-se que na descrição anterior, todo vértice  $z$  não alcançável de  $v$  não será incluído em  $Q$ . Desse modo, apenas os vértices e arestas alcançáveis da raiz são incluídos na busca.

O Algoritmo 4.4, recursivo, implementa a ideia.

---

**Algoritmo 4.4:** Busca em profundidade em digrafos

---

**Dados:** digrafo  $D(V, E)$

**Procedimento**  $P(v)$

```

    marcar  $v$ 
    colocar  $v$  na pilha  $Q$ 
    para  $w \in A(v)$  efetuar
        visitar  $(v, w)$ 
        se  $w$  não marcado então
             $P(w)$ 
        retirar  $v$  de  $Q$ 
        desmarcar todos os vértices
        definir uma pilha  $Q$ 
        definir uma raiz  $s \in V$ 
     $P(s)$ 
```

---

Observa-se que a pilha  $Q$  (a qual corresponde ao caminho  $Q$  da descrição dada) pode ser ignorada no Algoritmo 4.4, sem modificar a essência do procedimento. Contudo ela foi incluída para reforçar a ideia de que  $Q$  é implicitamente construída pela recursão.

A fim de examinar o efeito da busca, supõe-se de início que a raiz  $s$  seja também raiz do digrafo. Considere agora a visita a uma aresta  $(v, w)$ . Seja  $v$  alcançado antes de  $w$  na busca. Se  $w$  se encontrava desmarcado no momento da visita, então  $(v, w)$  é chamada *aresta de árvore*, caso contrário ( $w$  marcado) é denominado *aresta de avanço*. Seja agora  $w$  alcançado antes de  $v$  na busca. Se  $w \in Q$  no momento da visita, então  $(v, w)$  denomina-se *aresta de retorno*, caso contrário,  $w \notin Q$ , e a aresta  $(v, w)$  é denominada *aresta de cruzamento*. Desse modo, a busca em profundidade em um digrafo divide o seu conjunto de arestas em quatro subconjuntos disjuntos.

Seja  $D(V, E)$  um digrafo de raiz  $s$ . Considere uma busca em profundidade, de raiz também  $s$ , efetuada em  $D$ . Seja  $E_T$  o conjunto das arestas de árvore, obtido pela busca. Então analogamente ao caso não direcionado, pode-se mostrar que  $(V, E_T)$  é uma árvore direcionada enraizada geradora do digrafo  $D$ . Essa árvore é denominada *árvore de profundidade de raiz*  $s$ , do digrafo  $D$ .

#### Teorema 4.4

Seja  $D(V, E)$  um digrafo de raiz  $s$  e  $T(V, E_T)$  uma árvore de profundidade de raiz  $s$ , obtida a partir de uma busca em profundidade em  $D$ . Então toda aresta de árvore  $(v, w)$  é tal que  $v$  é pai de  $w$  em  $T$ ; toda aresta de avanço  $(v, w)$  é tal que  $v$  é ancestral, mas não pai, de  $w$  em  $T$ ; toda aresta de retorno  $(v, w)$  é tal que  $v$  é descendente de  $w$  em  $T$ ; toda aresta de cruzamento  $(v, w)$  é tal que  $v$  não é ancestral nem descendente de  $w$ , em  $T$ .

**Prova** Considere uma aresta  $(v, w) \in E$ . Suponha inicialmente  $v$  alcançado antes de  $w$  na busca. Como  $v$  atinge  $w$  em  $D$ , garante-se que  $w$  será alcançado na busca antes de se retirar  $v$  de  $Q$ , ou seja,  $v$  é ancestral de  $w$  em  $T$ . Se além disso,  $w$  se encontrava desmarcado quando alcançado, a computação  $P(w)$  é iniciada pelo procedimento  $P(v)$ , o que garante que  $v$  é pai de  $w$ , em  $T$ . Suponha agora  $w$  alcançado antes de  $v$  na busca. Se  $w \in Q$  no momento em que  $v$  é alcançado, então quando  $(v, w)$  é visitado  $Q$  contém  $s, \dots, w, \dots, v$ , ou seja,  $v$  é descendente de  $w$  em  $T$ . Caso contrário ( $w \notin Q$  no momento considerado),  $v$  não pode ser descendente de  $w$  em  $T$  nem tampouco ancestral, pois isto contradiria  $w$  alcançado antes de  $v$  na busca. ■

Seja  $D(V, E)$  um digrafo de raiz  $s$ . Numa busca em profundidade de raiz  $s$  em  $D$ , cada aresta de  $D$  é examinada exatamente uma vez. Conclui-se então que o processo de busca em profundidade para digrafos possui também complexidade  $O(n + m)$ .

Como exemplo, a Figura 4.7(b) corresponde a uma busca em profundidade de raiz  $s$ , no digrafo da Figura 4.7(a). Supõe-se que para esta busca, seja utilizada a seguinte ordenação das listas de adjacência do digrafo:

$$\begin{array}{ll} A(s) = \langle a, g, b \rangle & A(a) = \langle c \rangle \\ A(b) = \emptyset & A(c) = \langle b, d \rangle \\ A(d) = \langle a, e \rangle & A(e) = \langle c \rangle \\ A(f) = \langle e, g \rangle & A(g) = \langle f, h \rangle \\ A(h) = \langle i, d \rangle & A(i) = \emptyset \end{array}$$

Assim como no caso não direcionado, a ordem em que os vértices de um digrafo  $D(V, E)$  são incluídos e excluídos da pilha  $Q$ , em uma busca em profundidade em  $D$ , é importante para certas aplicações. Assim sendo, para cada vértice  $v \in V$  define-se *profundidade de entrada de  $v$* ,  $\text{PE}(v)$ , e *profundidade de saída de  $v$* ,  $\text{PS}(v)$ , respectivamente, como sendo o número de ordem em que  $v$  foi incluído e excluído da pilha  $Q$ . Para a busca da Figura 4.7(b) e considerando-se as listas de adjacência como dispostas anteriormente, seriam as seguintes as profundidades dos vértices do digrafo da Figura 4.7(a).

$v$	$s$	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
$\text{PE}(v)$	1	2	4	3	5	6	8	7	9	10
$\text{PS}(v)$	10	5	1	4	3	2	6	9	8	7

Observa-se que a sequência dos vértices  $v$  de um digrafo  $D$  em ordem crescente de  $\text{PE}(v)$  corresponde a um caminhamento preordem na árvore de profundidade produzida pela busca correspondente. Os valores das profundidades de entrada e saída dos vértices podem ser utilizados para classificar as arestas de um digrafo através de uma busca em profundidade, utilizando-se o seguinte lema.

#### Lema 4.3

Seja  $D(V, E)$  um digrafo de raiz  $s$  e  $(v, w) \in E$  uma aresta de  $D$ . Seja  $B$  uma busca em profundidade de raiz  $s$ , em  $D$ . Então:

- (i)  $(v, w)$  é aresta de árvore ou avanço se e somente se  $\text{PE}(v) < \text{PE}(w)$
- (ii)  $(v, w)$  é aresta de retorno se e somente se  $\text{PE}(v) > \text{PE}(w)$  e  $\text{PS}(v) < \text{PS}(w)$
- (iii)  $(v, w)$  é aresta de cruzamento se e somente se  $\text{PE}(v) > \text{PE}(w)$  e  $\text{PS}(v) > \text{PS}(w)$

**Prova** Aplicar a definição. ■

Seja  $(v, w)$  uma aresta do digrafo  $D$ , e considere uma busca em profundidade em  $D$ . Pela definição, sabe-se que  $(v, w)$  é aresta de árvore se e somente se  $w$  estiver desmarcado quando  $(v, w)$  foi visitado. Esta observação juntamente com o Lema 4.3 corresponde a um processo eficiente para se classificar as arestas de  $D$ , em arestas de árvore, avanço, retorno e cruzamento. A complexidade deste processo é  $O(n + m)$ .

Considere, novamente, um digrafo  $D$  e uma busca em profundidade de raiz  $s_1$ , em  $D$ . Seja agora o caso em que  $s_1$  não é raiz de  $D$ . Utilizando-se o Algoritmo 4.4, quando  $s_1$  fosse retirado da pilha  $Q$  (ou seja, ao término do algoritmo), a árvore definida pela busca não conteria todos os vértices de  $D$ . A árvore compreenderia exatamente o subconjunto de vértices alcançáveis de  $s_1$  em  $D$ . Para que cada vértice de  $D$  seja incluído na busca, torna-se necessário definir uma nova raiz  $s_2$ , não alcançável de  $s_1$ , e repetir o processo. Observe que na busca de raiz  $s_2$  toda aresta do tipo  $(v, w)$  em que  $w$  seja alcançável de  $s_1$  corresponderá a uma aresta de cruzamento da árvore de raiz  $s_2$  para a de raiz  $s_1$ . E assim por diante. Ao término de cada busca, verifica-se se todos os vértices do digrafo foram incluídos no processo. Em caso positivo, a *busca completa* é dada por encerrada. Caso contrário, uma nova busca é efetuada. Cada uma dessas produz uma árvore de profundidade. O conjunto das árvores obtidas ao longo do processo é chamado *floresta de profundidade*. Observe que se  $T'$  e  $T''$  são duas árvores da floresta tais que

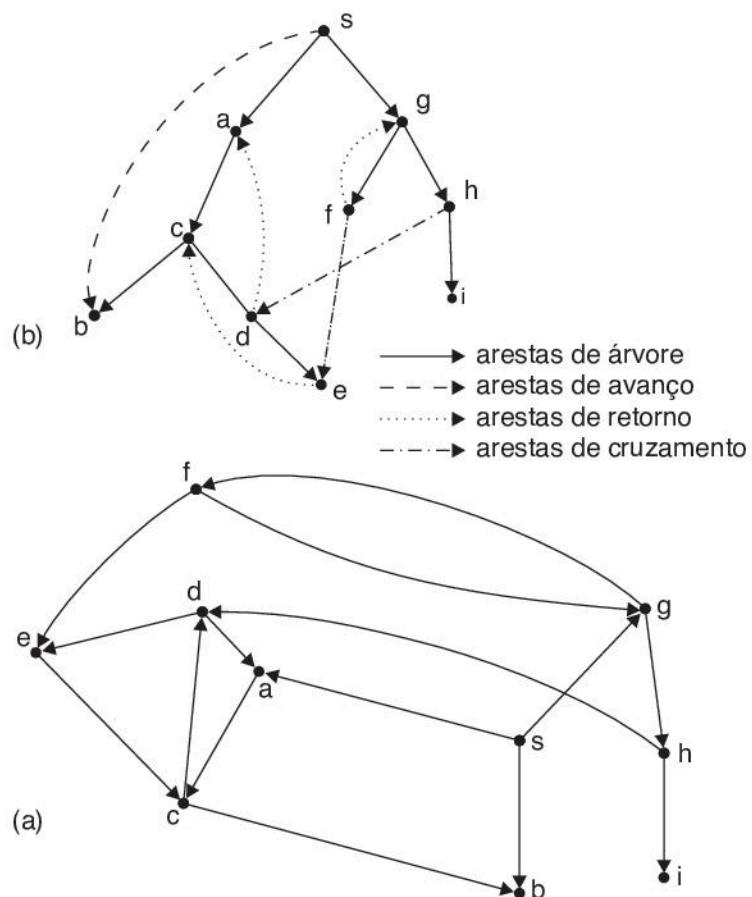


Figura 4.7: Busca em profundidade em digrafos

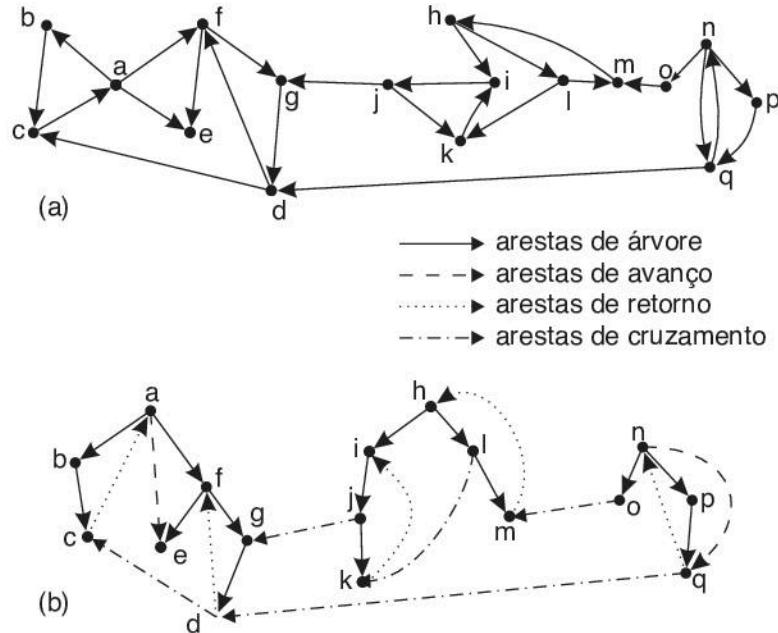


Figura 4.8: Busca em profundidade completa em digrafos. Caso geral

existe aresta de cruzamento de um vértice  $v'$  de  $T'$  para outro  $v''$  de  $T''$ , então  $T''$  foi construída antes de  $T'$  pelo algoritmo.

Note que para efetuar uma busca em profundidade completa pode-se utilizar o mesmo procedimento recursivo  $P(v)$  do Algoritmo 4.4. Basta substituir a chamada  $P(s)$ , da última linha desse algoritmo por:

**para**  $s \in V$  **efetuar**

se  $s$  é não marcado **então**

$P(s)$

A Figura 4.8(b) apresenta uma busca completa realizada no digrafo da Figura 4.8(a). A busca produziu uma floresta de profundidade composta por três árvores, com raízes  $a$ ,  $h$  e  $n$ , respectivamente.

## 4.6 Componentes Fortemente Conexas

Nesta seção apresenta-se um algoritmo para a determinação das componentes fortemente conexas de um digrafo, o qual ilustra uma aplicação de busca em profundidade.

### Lema 4.4

Seja  $D(V, E)$  um digrafo,  $T(V, E_T)$  uma floresta de profundidade de  $D$  e  $S(V_S, E_S)$  uma componente fortemente conexa de  $D$ . Então os vértices de  $V_S$  constituem uma subárvore parcial de  $T$ .

**Prova** Seja  $r$  o primeiro vértice de  $V_S$  a ser alcançado na busca correspondente a  $T$ . Então os vértices de  $V_S$  se encontram em  $T$  na subárvore de raiz  $r$ . Para completar a prova, basta mostrar que se  $w \in V_S$ ,  $w \neq r$ , e  $v$  é o pai de  $w$  em  $T$  então necessariamente  $v \in V_S$ . Mas o fato é evidente, pois as condições apresentadas implicam a existência de caminho em  $D$ , tanto de  $r$  a  $v$  (usando a árvore) como de  $v$  a  $r$  (usando a aresta  $(v, w)$  seguida do caminho de  $w$  a  $r$  em  $D$ ). ■

Pelo Lema 4.4, conclui-se que as componentes fortemente conexas de  $D$  partitionam a floresta de profundidade  $T'$  em subárvores disjuntas. Portanto, se for possível identificar as raízes dessas subárvores, as componentes ficam automaticamente determinados. Essas raízes serão denominadas *vértices fortes* de  $D$ .

Seja  $D(V, E)$  um digrafo e  $T(V, E_T)$  uma floresta de profundidade de  $D$ . Define-se a função  $\text{low}: V \rightarrow \{1, \dots, n\}$ , do seguinte modo. Para cada  $v \in V$ ,  $\text{low}(v) = \text{PE}(z)$ , onde  $z$  é o vértice de menor profundidade

de entrada localizado na mesma componente fortemente conexa do que  $v$ , que pode ser alcançado a partir de  $v$ , através de zero ou mais arestas de árvore seguidas por, no máximo, uma aresta de retorno ou cruzamento.

Como exemplo, o digrafo e a busca da Figura 4.8 forneceriam os seguintes valores para a função low.

$v$	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$	$j$	$k$	$l$	$m$	$n$	$o$	$p$	$q$
$PE(v)$	1	2	3	7	5	4	6	8	9	10	11	12	13	14	15	16	17
$low(v)$	1	1	1	3	5	3	3	8	9	9	9	8	8	14	15	14	14

A aplicação da função low na determinação das componentes fortemente conexas se dá através do Lema 4.5.

### Lema 4.5

Seja  $D(V, E)$  um digrafo. Um vértice  $v \in V$  é forte se e somente se  $PE(v) = low(v)$ .

**Prova** Obviamente,  $low(v) \leq PE(v)$ , para todo  $v \in V$ . Além disso, a igualdade deve existir para pelo menos um vértice de cada componente. Se  $v$  é forte, pelo Lema 4.4  $v$  possui a menor profundidade da entrada dentre os vértices de seu componente. Logo,  $low(v) = PE(v)$ . Reciprocamente, se  $low(v) = PE(v)$  não pode haver caminho em  $D$  de um descendente de  $v$  para algum seu ancestral. Logo,  $v$  é forte. ■

Observe que a utilização da função low, na determinação das componentes fortemente conexas de um digrafo, aparentemente apresenta uma circularidade. Para se obter as componentes é necessário previamente identificar os vértices fortes. Para essa identificação, utiliza-se a função low. Para computar essa função, seria necessário conhecer se dois dados vértices pertencem ou não a um mesmo componente.

Na realidade, a circularidade apresentada é de fato apenas aparente. O cálculo da função low pode ser realizado simultaneamente à determinação das componentes fortemente conexas de  $D$ , através de uma busca em profundidade. Ambos podem ser obtidos a partir da floresta de profundidade  $T$ , de baixo para cima durante a busca. Quando um vértice é alcançado pela primeira vez no processo, define-se  $low(v) := PE(v)$ . Este valor será atualizado no decorrer da computação, se necessário. Ao final da exploração de  $v$ ,  $low(v)$  terá sido calculado corretamente. Neste momento, se  $low(v) = PE(v)$  então  $v$  é forte. Nesse caso retiram-se de  $T$  todos os descendentes de  $v$ , os quais induzem em  $D$  uma componente fortemente conexa de raiz  $v$ . Caso contrário, se  $low(v) \neq PE(v)$  então a raiz  $r$  do componente ao qual  $v$  pertence é um ancestral próprio de  $v$  em  $T$ . Isto significa que  $v$  permanecerá em  $T$  até o final da exploração de  $r$ . Para efeito de cálculo de  $low(v)$ , considere a visita à aresta  $(v, w)$  durante a exploração de  $v$ . Os casos seguintes podem ocorrer:

- (i)  $(v, w)$  é aresta de árvore: Após a exploração de  $w$ , será conhecido o valor  $low(w)$ . Então se  $low(w) < low(v)$ , deve ser feita a atribuição  $low(v) := low(w)$ .
- (ii)  $(v, w)$  é aresta de avanço: As arestas de avanço não alteram as componentes fortemente conexas de um digrafo. Podem, portanto, ser ignoradas.
- (iii)  $(v, w)$  é aresta de retorno: Nesse caso  $v, w$  pertencem necessariamente à mesma componente fortemente conexa. Então, se  $PE(w) < low(v)$  efetua-se  $low(v) := PE(w)$ .
- (iv)  $(v, w)$  é aresta de cruzamento: Surge o problema de determinar se  $v$  pertence ou não à componente fortemente conexa  $F$  ao qual  $w$  faz parte. É imediato verificar que a resposta a esta questão será afirmativa precisamente quando o vértice forte  $z$  de  $F$  for um ancestral próprio de  $v$  em  $T$ . Nesse caso, a exploração de  $z$  ainda não foi completada, pois a de  $v$  também não o foi. Consequentemente os vértices de  $F$  ainda não foram retirados de  $T$ . Portanto,  $v, w$  pertencem a um mesmo componente se e somente se  $w$  se encontra em  $T$  no momento da visita à aresta  $(v, w)$ . Sendo essa condição satisfeita, se  $PE(w) < low(v)$ , efetua-se  $low(v) := PE(w)$ .

Essas observações conduzem ao Algoritmo 4.5.

O algoritmo pode ser implementado sem dificuldade em tempo  $O(n + m)$ . Como exemplo, na busca da Figura 4.8(b) correspondente ao digrafo 4.8(a), o primeiro vértice a ser identificado como forte é  $e$ , cujo único descendente em  $T$  é ele mesmo. Após a sua retirada de  $T$ , identifica-se  $a$  como forte, o que produz a retirada de  $T$  dos vértices  $\{a, b, c, d, f, g\}$ . O próximo vértice forte encontrado é  $i$ , correspondendo ao componente  $\{i, j, k\}$ . Em seguida  $h$  é detetado como forte, sendo  $\{h, l, m\}$  seus descendentes a serem retirados de  $T$ . O vértice forte

---

**Algoritmo 4.5:** Componentes fortemente conexas de um digrafo**Dados:** digrafo  $D(V, E)$ **Procedimento**  $F(v)$     marcar  $v$      $j := j + 1$      $\text{PE}(v) := \text{low}(v) := j$     **para**  $w \in A(v)$  **efetuar**        **se**  $w$  não marcado **então**            incluir  $(v, w)$  em  $T$ , sendo  $v$  pai de  $w$              $F(w)$              $\text{low}(w) := \min\{\text{low}(v), \text{low}(w)\}$         **caso contrário**            **se**  $w$  está em  $T$  **então**                 $\text{low}(v) := \min\{\text{low}(v), \text{PE}(w)\}$             **se**  $\text{low}(v) = \text{PE}(v)$  **então**                retirar  $v$  e seus descendentes de  $T$ 

▷ (induzem uma componente fortemente conexa)

 $j := 0$ 

desmarcar todos os vértices

    definir uma árvore  $T$ , vazia    **para**  $s \in V$  **efetuar**        **se**  $s$  não marcado **então**             $F(s)$ 

o dá origem a uma componente constituída apenas por  $o$  e o vértice  $n$  corresponde à componente  $\{n, p, q\}$ . A Figura 4.9 ilustra a saída produzida.

## 4.7 Busca em Largura

Na Seção 4.2 foi definido o conceito geral de busca. O Algoritmo 4.1 teve como finalidade efetuar uma busca geral em grafos não direcionados. Um critério específico de escolha de vértice marcado (Seção 4.3) deu origem à busca em profundidade. Um outro critério, descrito na presente seção, corresponde à busca em largura.

Uma busca é dita em *largura* quando o critério de escolha de vértice marcado obedecer a:

“entre todos os vértices marcados e incidentes a alguma aresta ainda não explorada, escolher aquele *menos recentemente alcançado na busca*”.

Assim como a busca em profundidade pode ser implementada com o auxílio de uma pilha, a busca em largura utiliza uma fila. De fato, pode-se formular para esses dois casos algoritmos que sejam absolutamente idênticos, a menos da estrutura auxiliar utilizada (pilha ou fila).

O algoritmo seguinte implementa essa busca.

À semelhança da busca em profundidade, o critério de escolha de arestas também é arbitrário nesse caso. Também no Algoritmo 4.6 esse critério é dado pela ordenação de  $A(v)$ .

As visitas a arestas são realizadas nas linhas (I) e (II) apresentadas anteriormente. Seja  $E_T$  o conjunto das arestas visitadas em (I).

### Teorema 4.5

O grafo  $T(V, E_T)$  é uma árvore geradora de  $G$ .

A prova é semelhante à do Teorema 4.1. A árvore  $T(V, E_T)$  recebe o nome de *árvore de largura* de  $G$ . A árvore  $T$  será considerada enraizada, sendo sua raiz precisamente o vértice raiz da busca. Denomina-se  $\text{nível}(v)$  o *nível* do vértice  $v$  em  $T$ .

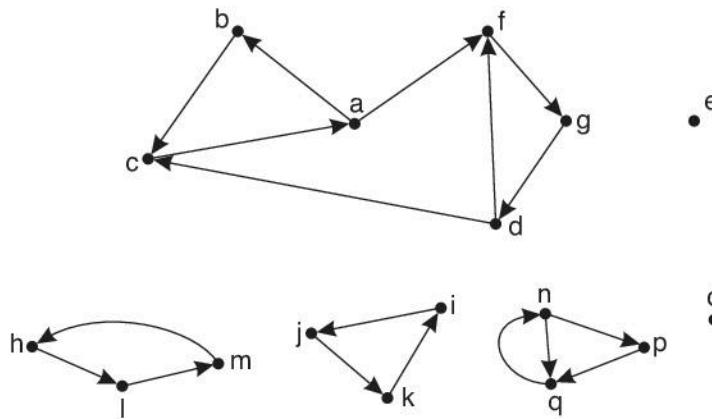


Figura 4.9: Componentes fortemente conexas do digrafo da Figura 4.8(a)

**Algoritmo 4.6:** Busca em largura**Dados:** grafo  $G(V, E)$ , conexo

desmarcar todos os vértices

escolher uma raiz  $s \in V$ definir uma fila  $Q$ , vaziamarcar  $s$ inserir  $s$  em  $Q$ **enquanto**  $Q \neq \emptyset$  **efetuar**  seja  $v$  o primeiro elemento de  $Q$   **para**  $w \in A(v)$  **efetuar**    se  $w$  é não marcado **então**      visitar  $(v, w)$       marcar  $w$       inserir  $w$  em  $Q$   **caso contrário**    se  $w \in Q$  **então**      visitar  $(v, w)$     retirar  $v$  de  $Q$ 

▷ (I)

▷ (II)

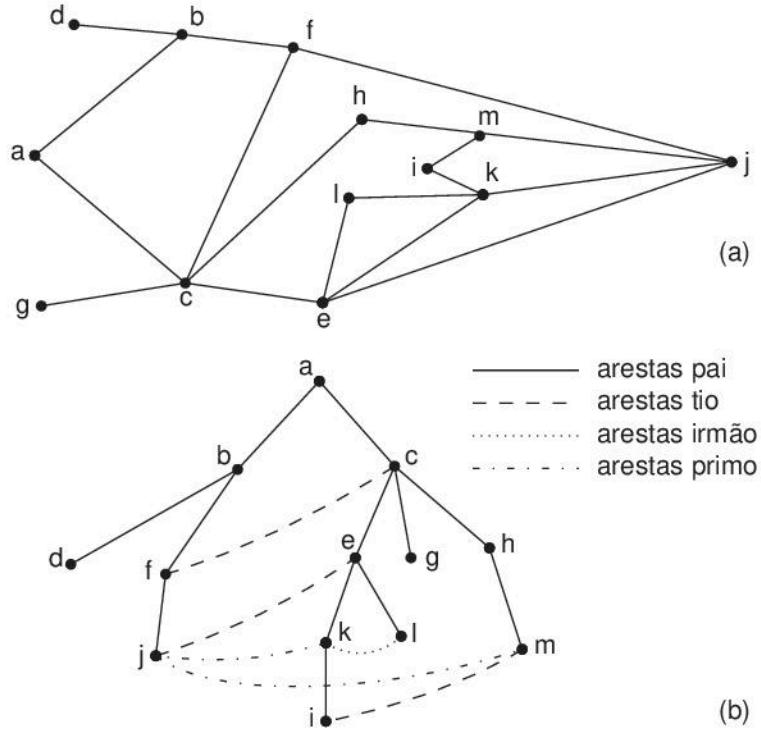


Figura 4.10: Tipos de arestas na busca em largura

**Teorema 4.6**

Seja  $G(V, E)$  um grafo conexo,  $(v, w) \in E$  e  $T(V, E_T)$  uma árvore de largura de  $G$ . Então  $\text{nível}(v) \neq \text{nível}(w)$  diferem, no máximo, de uma unidade.

**Prova** Seja  $v$  alcançado antes de  $w$  na busca em largura correspondente à árvore  $T$ . Então  $\text{nível}(v) \leq \text{nível}(w)$ . Agora se  $\text{nível}(v) \neq \text{nível}(w)$  diferem de mais de uma unidade, então  $w \in A(v)$  não foi alcançado quando a lista  $A(v)$  foi examinada, o que é absurdo. ■

Observe que no Algoritmo 4.6, cada aresta  $(v, w)$  do grafo é examinada exatamente duas vezes, uma para  $v \in A(w)$  e outra para  $w \in A(v)$ . Como consequência, tem-se que a complexidade da busca em largura também é linear no tamanho do grafo.

Observa-se também que o Teorema 4.6 divide naturalmente o conjunto de arestas de  $G$  em duas partes. Arestas  $(v, w)$  tais que  $|\text{nível}(v) - \text{nível}(w)| = 1$  e arestas em que  $|\text{nível}(v) - \text{nível}(w)| = 0$ . Cada uma dessas partes pode ser ainda particionada em duas outras. Seja a aresta  $(v, w)$  com  $\text{nível}(v) \leq \text{nível}(w)$ :

- (i)  $(v, w)$  é aresta pai (ou aresta de árvore), quando  $v$  é pai de  $w$  em  $T$ , ou seja  $(v, w) \in E_T$ .
- (ii)  $(v, w)$  é aresta tio quando  $\text{nível}(w) = \text{nível}(v) + 1$  e  $(v, w) \notin E_T$ .
- (iii)  $(v, w)$  é aresta irmão quando  $v$  e  $w$  possuem o mesmo pai em  $T$ .
- (iv)  $(v, w)$  é aresta primo quando  $\text{nível}(v) = \text{nível}(w)$  e  $v, w$  não possuem o mesmo pai em  $T$ .

A Figura 4.10(b) apresenta uma busca em largura realizada no grafo da Figura 4.10(a).

A ordem em que os vértices são retirados da fila  $Q$  pode ser relevante para algumas aplicações. Para cada  $v \in V$  e para uma dada busca em largura, define-se *largura* de  $(v)$  como sendo o número de ordem em que  $v$  foi retirado da fila  $Q$ . Supondo que a ordenação das listas de adjacência do grafo da Figura 4.10(a) obedeça à ordem alfabética de seus rótulos, seriam as seguintes larguras dos vértices desse grafo, relativas à busca da Figura 4.10(b).

vértice $v$	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$	$j$	$k$	$l$	$m$
$\text{largura}(v)$	1	2	3	4	6	5	7	8	13	9	10	11	12

Finalmente, também nesse caso, para grafos desconexos aplica-se a busca em largura separadamente para cada uma de suas componentes conexas.

## 4.8 Busca em Largura Lexicográfica

Até o momento, foram examinados dois critérios diferentes para a seleção de vértice marcado em uma busca. Eles deram origem às buscas em profundidade e em largura, respectivamente. Em ambos os casos, a escolha de aresta incidente ao vértice marcado (isto é, da aresta a ser explorada) é arbitrária. Como decorrência, os algoritmos correspondentes são em geral sensíveis à ordenação dos vértices nas listas de adjacência. Nesta seção, examina-se um critério para seleção de aresta incidente, em uma busca em largura. O critério não é absoluto, no sentido de que não necessariamente conduz à seleção única da aresta. Contudo, sua aplicação pode tornar a busca menos dependente em relação à ordenação das listas de adjacências.

Considere uma busca em largura realizada em um grafo não direcionado conexo  $G(V, E)$ . Cada vértice  $v \in V$ , em algum instante do processo, é marcado e incluído numa fila  $Q$ , conforme descreve o Algoritmo 4.6. Na ocasião em que  $v$  é excluído da fila, acontece a exploração das arestas incidentes a  $v$ , cuja ordem de escolha é arbitrária (o passo correspondente no Algoritmo 4.5 é o bloco *para* definido na linha \*). Os vértices  $w \in A(v)$  marcados não são incluídos em  $Q$ . Por este motivo, a ordem em que são alcançados (isto é, a ordem em que as arestas  $(v, w)$  correspondentes são exploradas) não influí no formato da árvore de largura obtida na busca. Considere agora os vértices do conjunto  $A^*(v) = \{w \in A(v) | w \text{ é não marcado}\}$ . Cada  $w \in A^*(v)$  é incluído em  $Q$ .

A ordem relativa de sua inclusão (e portanto de sua *exclusão*) de  $Q$  corresponde à ordem da exploração das arestas  $(v, w)$  respectivas. O objetivo presente é descrever um critério para tentar distinguir, se possível, entre vértices de  $A^*(v)$  de modo a estabelecer uma ordem relativa de exploração dos mesmos. A ordem em que os vértices  $w \in A(v)$  marcados são alcançados no processo será considerada como irrelevante, para a presente situação.

Observe que a busca não se modifica se o problema da seleção de  $w \in A^*(v)$ , para o vértice marcado  $v$ , for adiado até a exclusão de  $w$  da fila  $Q$ . Em outras palavras, ao invés de efetuar a seleção para definir a ordem de entrada em  $Q$ , a ideia consiste em colocar todos os vértices do subconjunto  $A^*(v)$  em uma mesma posição em  $Q$ . Posteriormente, por ocasião da exclusão da fila, dos vértices de  $A^*(v)$ , aplica-se o critério de seleção. Considere então,  $w_1, w_2 \in A^*(v)$ , no momento da exclusão de  $Q$ . Diz-se que  $w_1$  é *mais remoto* do que  $w_2$  quando existir algum vértice marcado  $z$ , com  $(z, w_1) \in E$  mas  $(z, w_2) \notin E$ , tal que todo vértice marcado  $z'$ , com  $(z', w_1) \notin E$  mas  $(z', w_2) \in E$ , satisfaz  $\text{largura}(z) < \text{largura}(z')$ . Isto é, quando  $w_1$  é mais remoto do que  $w_2$ , existe um vértice  $z$  adjacente a  $w_1$  e não a  $w_2$ , que foi alcançado na busca antes de qualquer vértice  $z'$  adjacente a  $w_2$  e não a  $w_1$ . Observe que nesta definição, é irrelevante o efeito dos vértices simultaneamente adjacentes a  $w_1$  e  $w_2$ . Quando dois vértices são tais que cada um deles não é mais remoto do que o outro, diz-se que os mesmos são *igualmente remotos*. Como exemplo, considere a busca em largura representada na Figura 4.11. As arestas de árvore da busca correspondem às linhas retas do desenho e a numeração indicada é a largura de cada vértice. O subconjunto  $A^*(7)$  é  $\{11, 12, 13\}$ , pois o vértice 10 se encontrava marcado quando 7 ocupava a posição inicial da fila. Observe que 11 é mais remoto do que 12 bem como 6 mais do que 5. Os vértices 11 e 13 são igualmente remotos, assim como o são 7 e 8.

Diz-se que uma busca em largura em um grafo não direcionado  $G(V, E)$  é *lexicográfica* quando para todo  $v \in V$  e  $w_1, w_2 \in A^*(v)$ , se  $w_1$  é mais remoto do que  $w_2$  então  $w_1$  é explorado antes de  $w_2$  na busca. Assim sendo, a sequência em que os  $w \in A^*(v)$  são explorados corresponde à ordenação do mais para o menos remoto em  $A^*(v)$ . A ordem relativa dos igualmente remotos é arbitrária. Observe que essa ordem de exploração corresponde à sequência em que os vértices são retirados da fila utilizada na busca.

Uma busca em largura lexicográfica pode ser reconhecida através da árvore de largura correspondente e da posição das demais arestas. Como exemplo, a busca em largura representada pela Figura 4.12(a) é lexicográfica, enquanto que a da (b) não o é (pois o vértice 6, explorado obviamente após 5, é mais remoto do que este na 4.12(b)). Uma árvore de largura obtida através de uma busca lexicográfica é denominada *árvore de largura lexicográfica*.

A seguinte definição é útil para implementar este processo de busca. Sejam  $S_1 = x_1, \dots, x_p$  e  $S_2 = y_1, \dots, y_q$  duas sequências de inteiros. Diz-se que  $S_1$  é *lexicograficamente maior* do que  $S_2$ , denotando  $S_1 > S_2$ , quando:

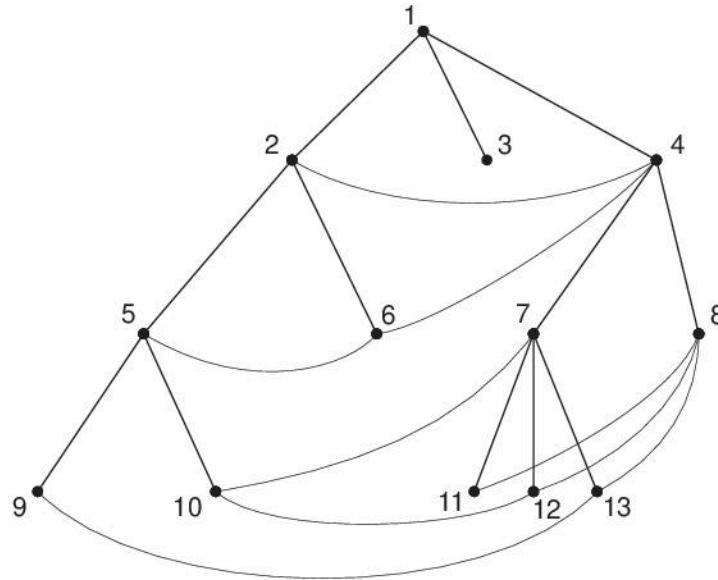


Figura 4.11: Uma busca em largura

- (i) Existe algum índice  $j$ ,  $1 \leq j \leq p, q$ , tal que  $x_j > y_j$ , e para todo  $k$ ,  $1 \leq k < j$ ,  $x_k = y_k$ . Ou então:
- (ii)  $p > q$  e para todo  $k$ ,  $1 \leq k \leq q$ ,  $x_k = y_k$ .

Assim, por exemplo,  $649 > 6489$  e  $3242 > 324$ . As sequências  $S_1, \dots, S_n$  estão em *ordenação lexicográfica crescente (decrescente)* quando  $S_i < S_j \Rightarrow i < j$  ( $S_i > S_j \Rightarrow i > j$ ). Por exemplo, as palavras em um dicionário estão organizadas segundo uma ordenação lexicográfica crescente, supondo  $A < B < \dots < Z$ .

Seja uma busca em largura em um grafo não direcionado  $G(V, E)$ . A ideia consiste em utilizar uma sequência de rótulos inteiros  $R(w)$  para cada  $w \in V$ , de modo a facilitar uma implementação eficiente do processo. Seja um processo de busca no qual  $w \in V$  é adjacente aos vértices já marcados,  $z_1, \dots, z_k$ , em ordem crescente de suas larguras. Então  $R(w)$  é a sequência dos complementos das larguras dos  $z_j$ . O *complemento de largura* de um vértice  $z$  é definido como  $n + 1 - \text{largura}(z)$ , sendo  $n = |V|$ . Cada  $R(w)$ , portanto, é uma sequência decrescente de inteiros. Por exemplo, na Figura 4.11, o vértice 3 é adjacente ao vértice 1, cujo complemento de largura é 13. Logo,  $R(3) = \{13\}$ . Da mesma forma, conclui-se que  $R(11) = \{7, 6, 5\}$ , na mesma figura.

Para implementar o processo de busca em largura lexicográfica é necessário desenvolver um critério simples que permita verificar, eficientemente para cada vértice  $v$ , qual o mais remoto dentre os  $w \in A * (v)$ . O lema seguinte responde a esta questão.

#### Lema 4.6

Seja uma busca em largura em um grafo não direcionado  $G(V, E)$ . Seja  $v \in V$  e  $w_1, w_2 \in A * (v)$ . Então  $w_1$  é mais remoto do que  $w_2$  se e somente se  $R(w_1) > R(w_2)$  lexicograficamente.

**Prova** Aplicar a definição. ■

Baseado no Lema 4.6, pode-se então descrever o algoritmo correspondente.

Observe que a fila  $Q$  bem como as visitas às arestas do grafo não foram explicitadas na descrição do algoritmo. Como exemplo, a Figura 4.13(b) representa uma busca em largura lexicográfica, no grafo da Figura 4.13(a). A sequência de rótulos  $R(w)$  está indicada para cada caso.

A rotulação  $R$  efetuada pelo Algoritmo 4.7 apresenta ainda a seguinte propriedade. Para cada  $v \in V$  e  $w_1, w_2 \in A * (v)$ , se num certo instante do processo de busca,  $R(w_1)$  se torna lexicograficamente maior do que  $R(w_2)$ , assim permanece até o final. Esta característica de monotonicidade é importante para justificar uma implementação simples do algoritmo em complexidade  $O(n + m)$ .

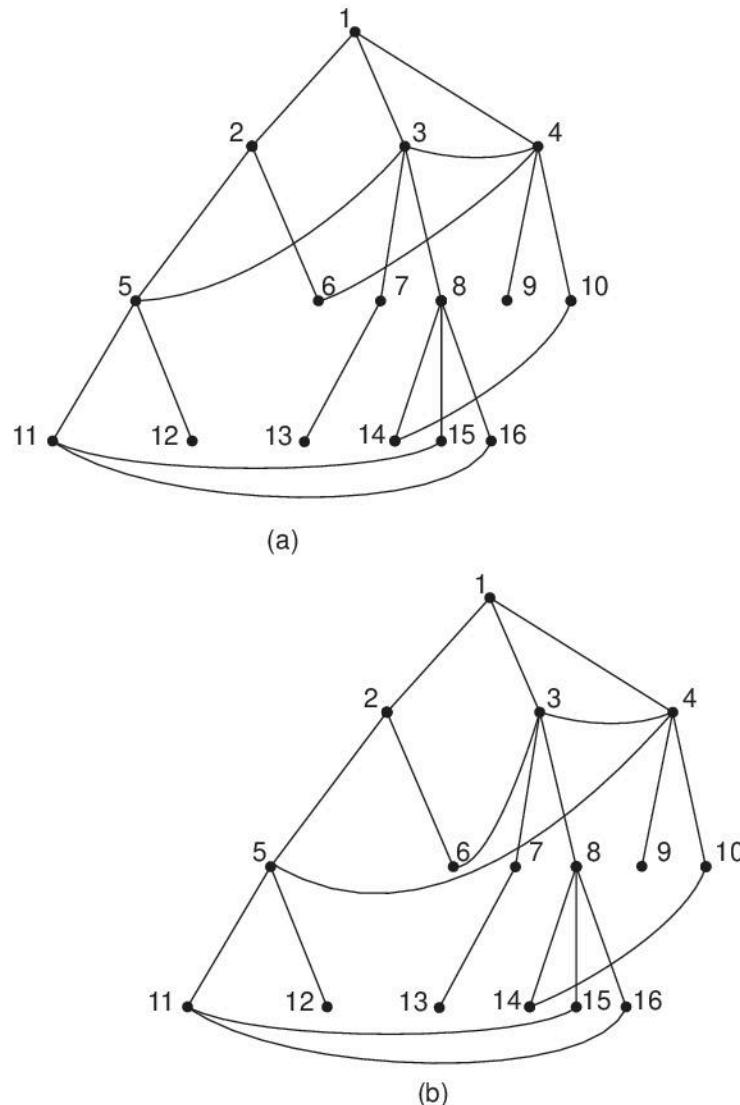


Figura 4.12: Busca em largura lexicográfica e não lexicográfica

**Algoritmo 4.7:** Busca em largura lexicográfica**Dados:** grafo  $G(V,E)$  conexo

desmarcar todos os vértices

**para**  $v \in V$ definir  $R(v) := \emptyset$ **para**  $j = n, n-1, \dots, 1$  **efetuar**escolher um vértice  $v$  não marcado com  $R(v)$  lexicograficamente máximomarcar  $v$ **para**  $w \in A(v)$  **efetuar e**  $w$  não marcado **efetuar**incluir  $j$  à direita em  $R(w)$

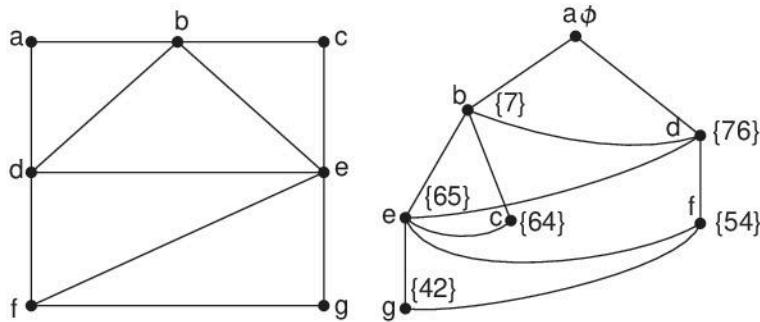


Figura 4.13: Busca em largura lexicográfica com rotulação de vértices

Finalmente, ressalte-se uma vez mais que não é absoluto o critério de seleção de arestas  $(v, w)$  incidente ao vértice marcado  $v$ , definido pela busca em largura lexicográfica. Com efeito, para  $w \in A^*(v)$ , o critério corresponde a escolher o  $w$  que maximiza  $R(w)$ . No caso de haver mais de um máximo, a escolha é arbitrária.

## 4.9 Reconhecimento dos Grafos Cordais

Descreve-se nessa seção uma aplicação de busca em largura lexicográfica.

Seja  $G(V, E)$  um grafo não direcionado e  $C$  um ciclo de  $G$ . Uma *corda* de  $G$  é uma aresta não pertencente a  $C$  e que une dois vértices de  $C$ . No grafo da Figura 4.14, a aresta  $(c, b)$  é uma corda do ciclo  $a, c, d, b, a$ . Um grafo é denominado *cordal* (ou *triangularizado*) quando todo ciclo de comprimento maior do que 3 possui uma corda. O grafo da Figura 4.13(a) é cordal, enquanto que o da Figura 4.14 não o é (pois o ciclo  $c, d, f, e, c$  de comprimento 4 não possui corda). Na presente seção apresenta-se um algoritmo para reconhecer grafos cordais.

Um vértice  $v \in V$  é denominado *simplicial* se o subgrafo induzido por  $A(v)$  for completo. Por exemplo, o vértice  $a$  do grafo da Figura 4.14 é simplicial. Aliás,  $a$  é o único vértice simplicial do exemplo. Seja agora um subconjunto  $S \subseteq V$ , e dois vértices não adjacentes  $v, w \in V$ . O subconjunto  $S$  é chamado  $v-w$  separador quando a remoção de  $G$  dos vértices em  $S$  produz um grafo em que  $v, w$  figuram em componentes conexas distintas. Ou seja,  $v, w$  pertencem a componentes conexas distintas de  $G - S$ . Se  $S$  não contiver subconjunto próprio que seja também um  $v-w$  separador, então  $S$  é denominado  $v-w$  separador minimal. No grafo da Figura 4.14,  $S = \{c, d\}$  é um  $b-e$  separador minimal, por exemplo.

As proposições seguintes são de interesse para o algoritmo.

### Lema 4.7

Seja  $G(V, E)$  um grafo cordal,  $|V| > 1$  e  $v \in V$ . Então o grafo  $G - v$  também é cordal.

A prova é imediata.

### Teorema 4.7

Seja  $G(V, E)$  um grafo não completo conexo. Então todo separador minimal induz um subgrafo completo se e somente se  $G$  for cordal.

**Prova** Para a prova de necessidade, se  $G$  for acíclico o teorema vale trivialmente. Suponha que  $G$  contém o ciclo  $v, t, w, z_1, \dots, z_k, v$ , com  $k \geq 1$ . Todo  $v-w$  separador minimal deve conter os vértices  $t$  e  $z_i$ , para algum  $i$ , logo  $(t, z_i) \in E$  e o ciclo contém uma corda. Para a prova de suficiência, seja  $S$  um  $v-w$  separador minimal e  $G_v, G_w$  as componentes conexas de  $G - S$  contendo  $v, w$  respectivamente (Figura 4.15). Cada  $s \in S$  é adjacente a vértices de  $G_v$  e  $G_w$ . Sejam  $s_1, s_2$  vértices distintos de  $S$ , sendo  $s_1, P_v, s_2$  e  $s_1, P_w, s_2$  os caminhos mínimos entre  $s_1$  e  $s_2$  contendo apenas vértices de  $G_v$  e  $G_w$ , à exceção dos extremos, respectivamente. Logo, o ciclo  $s_1, P_v, s_2, P_w, s_1$  possui comprimento  $\geq 4$  e por hipótese deve conter uma corda. Porque  $S$  é separador, não pode haver aresta entre vértices de  $P_v$  e  $P_w$ . Porque  $P_v, P_w$  são caminhos mínimos, essa corda não pode ligar vértices de  $P_v$  ou  $P_w$ , tanto entre si ou a  $s_1$  ou  $s_2$ . Então a única possibilidade é  $(s_1, s_2) \in E$ . Logo  $S$  induz um subgrafo completo. ■

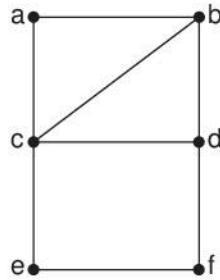


Figura 4.14: Grafo não cordal

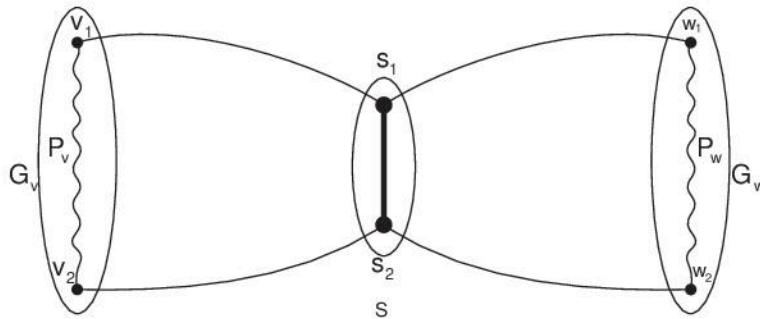


Figura 4.15: Prova do Teorema 4.7

**Teorema 4.8**

Seja  $G(V, E)$  um grafo cordal. Se  $G$  é completo, qualquer um de seus vértices é simplicial. Se  $G$  não é completo, ele contém um par de vértices simpliciais não adjacentes.

**Prova** Se  $G$  é completo a prova é imediata. Suponha que  $G$  contém dois vértices  $v, w$  não adjacentes e seja o lema verdadeiro para grafos com número de vértices  $< |V|$ . Seja  $S$  um  $v - w$  separador minimal e  $G_v, G_w$  as componentes conexas de  $G - S$  que contêm  $v, w$  respectivamente. Aplicando a hipótese de indução, conclui-se que (i)  $G_v + S$  é completo e todo vértice de  $G_v$  é simplicial em  $G_v + S$ , ou (ii)  $G_v + S$  possui 2 vértices simpliciais não adjacentes, com um deles pelo menos em  $G_v$  (pois pelo Teorema 4.7,  $S$  induz um subgrafo completo). Em qualquer caso,  $G_v$  contém um vértice simplicial em  $G_v + S$ , o qual necessariamente é simplicial em  $G$ . Por um raciocínio análogo, conclui-se que  $G_w$  contém um vértice, o qual é simplicial também em  $G$ . ■

O Teorema 4.8 é a base para a construção de um algoritmo para reconhecer grafos cordais. Dado o grafo  $G$ , inicialmente, localizar um vértice simplicial  $v$ . Excluir  $v$  de  $G$ . Pelo Lema 4.7, se  $G$  for cordal,  $G - v$  também o será. Repetir o processo até que não hajam mais vértices a serem retirados, ou então  $G$  não contenha vértice simplicial. Neste último caso,  $G$  não é cordal.

A fim de que o processo anterior possa ser aplicado, restam ainda duas tarefas importantes a serem realizadas:

- (i) Provar que se o processo anterior esgotar todos os vértices de  $G$ , então o grafo é necessariamente cordal.
- (ii) Encontrar uma forma eficiente de localizar vértices simpliciais.

Para a realização das tarefas anteriores, o seguinte conceito é importante. Seja  $G(V, E)$  um grafo,  $\alpha = v_1, v_2, \dots, v_n$  uma ordenação de seus vértices. A sequência  $\alpha$  é denominada *esquema de eliminação perfeita*, se cada  $v_i$  for um vértice simplicial do subgrafo induzido por  $\{v_i, v_{i+1}, \dots, v_n\}$ . Isto é, para cada  $v_i$ , o subgrafo induzido por  $A(v_i) - \{v_1, v_2, \dots, v_{i-1}\}$  é completo. Por exemplo, a sequência  $\alpha = a, c, b, d, e, f, g$  é um esquema de eliminação perfeita para o grafo da Figura 4.13(a). Naturalmente, essa sequência não é necessariamente única. Um outro esquema possível para este mesmo grafo é  $c, g, f, a, b, d, e$ . Por outro lado, o grafo da Figura 4.14 não admite esquema de eliminação perfeita. Pois o seu primeiro vértice seria  $a$ , o segundo  $b$ , mas não é possível determinar o terceiro. A utilidade desse conceito para grafos cordais provém do Teorema 4.9.

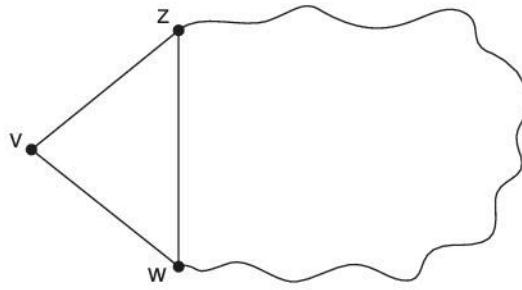


Figura 4.16: Prova do Teorema 4.9

**Teorema 4.9**

Um grafo  $G(V, E)$  é cordal se e somente se  $G$  possuir um esquema de eliminação perfeita.

**Prova** Se  $G$  é cordal, o algoritmo iterativo produzirá um esquema de eliminação perfeita. Reciprocamente, suponha que  $G$  possui um esquema de eliminação perfeita. Seja  $C$  um ciclo de  $G$  de comprimento  $> 3$  e  $v$  o vértice de  $C$ , mais à esquerda no esquema. Então  $A(v)$  contém pelo menos dois vértices  $w, z$  não consecutivos em  $C$  (Figura 4.16). Logo, como  $v$  é simplicial no subgrafo induzido por  $v$  e pelos vértices à direita de  $v$  no esquema, existe a aresta  $(w, z)$  que é uma corda de  $C$ . ■

O Teorema 4.9 responde à questão (i) anterior. Sabe-se agora que o problema de reconhecer se um dado grafo é cordal é equivalente ao problema de encontrar um esquema de eliminação perfeita do grafo. Mas como encontrar tal esquema, de forma eficiente? Recorre-se, então, à busca em largura lexicográfica da seção anterior. De fato, verifica-se que um esquema de eliminação perfeita corresponde a uma ordenação decrescente dos vértices  $v$ , segundo os números largura( $v$ ) definidos por uma busca em largura lexicográfica. Por exemplo, a busca da Figura 4.13(b) efetuada no grafo cordal da 4.13(a) encontraria o esquema de eliminação perfeita  $g, f, c, e, d, b, a$ . Observe que esta sequência pode ser obtida percorrendo-se a árvore de largura de baixo para cima e da direita para a esquerda. O lema seguinte atesta a correção do processo.

**Lema 4.8**

Seja  $G(V, E)$  um grafo cordal aplicado ao Algoritmo 4.7 de busca em largura lexicográfica. Então a sequência  $S$  de vértices  $v$  ordenados decrescentemente segundo largura( $v$ ) é um esquema de eliminação perfeita.

**Prova** Por definição de largura( $v$ ), a sequência  $S$  corresponde à ordem inversa em que os vértices de  $G$  são retirados da fila pelo Algoritmo 4.7. Suponha que  $S$  não seja um esquema de eliminação perfeita. Então, existem vértices  $v_1, v_2, v_3 \in V$ , tais que  $v_2, v_3 \in A(v_1)$ ,  $(v_2, v_3) \notin E$ , e  $v_2, v_3$  estão à direita de  $v_1$  em  $S$  (Figura 4.17(a)). Então, pela definição de busca em largura lexicográfica existe necessariamente um vértice  $v_4 \in V$ , à direita de  $v_3$  em  $S$ , tal que  $(v_2, v_4) \in E$ , mas  $(v_1, v_4) \notin E$  (Figura 4.17(b)). Sem perda de generalidade, seja  $v_4$  o vértice mais à direita em  $S$ , nessas condições. Então,  $(v_3, v_4) \notin E$ , caso contrário  $G$  não seria cordal. Pela definição de busca em largura lexicográfica, existe um vértice  $v_5 \in V$ , à direita de  $v_4$  em  $S$ , tal que  $(v_3, v_5) \in E$  (Figura 4.17(c)). Sem perda de generalidade, seja  $v_5$  o vértice mais à direita em  $S$ , nessas condições. Por outro lado,  $(v_4, v_5) \notin E$ , caso contrário  $G$  não seria cordal. Novamente existe um vértice  $v_6 \in V$ , à direita de  $v_5$  em  $S$ , com  $(v_4, v_6) \in E$ , e assim por diante. A situação se repetiria indefinidamente. Mas  $V$  é finito, uma contradição. Logo, necessariamente  $S$  é um esquema de eliminação perfeita. ■

Um processo para reconhecer grafos cordais pode ser então formulado. Dado um grafo  $G$ , aplica-se o Algoritmo 4.7, o qual produz como saída uma sequência  $S$  dos vértices  $v$  de  $G$ , ordenados decrescentemente segundo largura( $v$ ). Se  $G$  é cordal, o Lema 4.8 afirma que  $S$  é um esquema de eliminação perfeita. Caso contrário,  $G$  não é cordal, o Teorema 4.9 permite concluir que  $S$  não é tal esquema. Observa-se nesse ponto que há necessidade de utilizar um algoritmo que seja capaz de reconhecer se uma dada sequência  $S$  é ou não um esquema de eliminação perfeita. Portanto,  $G$  será cordal precisamente quando  $S$  for reconhecido como esquema.

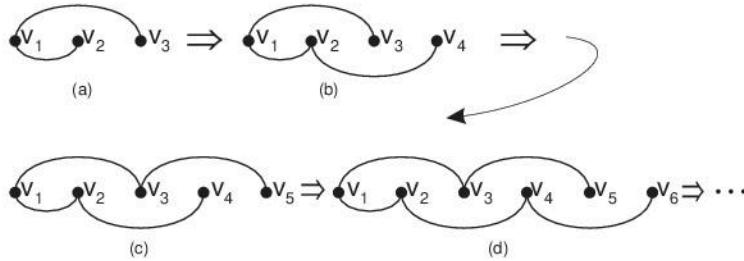


Figura 4.17: Prova do Lema 4.10

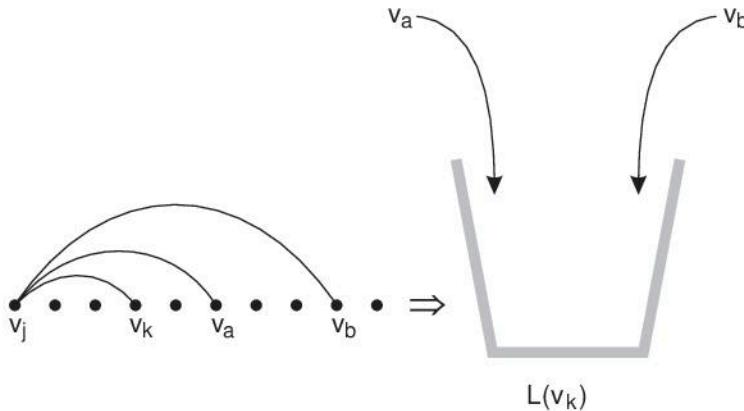


Figura 4.18: Reconhecimento de esquemas de eliminação perfeita

Esse reconhecimento pode ser realizado em tempo polinomial, simplesmente aplicando a definição. Dados  $G(V, E)$  e a sequência  $S = v_1, \dots, v_n$ , para cada  $v_i$  deve ser verificado se os vértices  $v_j \in A(v_i)$ , com  $j > i$ , induzem uma clique. Naturalmente,  $S$  é um esquema de eliminação perfeita se e somente se essas verificações forem todas satisfeitas. A aplicação direta desta estratégia conduz a um algoritmo de complexidade  $O(mn)$ . O seguinte processo alternativo é mais eficiente.

Considere a sequência  $S = v_1, \dots, v_n$  de vértices de  $G(V, E)$ , ordenada decrescentemente segundo valores de  $\text{largura}(v)$ , obtida como saída de uma busca em largura lexicográfica. Para cada vértice  $v_i \in V$ , define-se o multiconjunto  $L(v_i)$ , inicialmente vazio. A ideia consiste em percorrer  $S$  da esquerda para a direita. Para cada  $j$ ,  $1 \leq j \leq n$ , define-se o subconjunto  $A'(v_j) \subseteq A(v_j)$  contendo os vértices adjacentes e à direita de  $v_j$  em  $S$ . Se  $A'(v_j) \neq \emptyset$  então (i) determina-se o vértice  $v_k \in A'(v_j)$  mais próximo de  $v_j$  em  $S$ , e (ii) adicionam-se os vértices de  $A'(v_j) - \{v_k\}$  a  $L(v_k)$  (Figura 4.18). Ao final do processo, determina-se  $L(v_j) - A(v_j)$ , para todo  $j$ . Logo,  $S$  é um esquema de eliminação perfeita se e somente se a igualdade  $L(v_j) - A(v_j) = \emptyset$  for verdadeira para  $j = 1, \dots, n$ .

Como exemplo, a busca em largura lexicográfica do grafo da Figura 4.13(a) forneceria a sequência  $g, f, c, e, d, b, a$ , com os seguintes valores para  $L(v_i)$ :

$$\begin{array}{ll} L(g) = \emptyset & L(e) = \{d, b\} \\ L(f) = \{e\} & L(d) = \{b\} \\ L(c) = \emptyset & L(b) = \{a\} \\ L(a) = \emptyset & \end{array}$$

Observe que  $L(v_i) - A(v_i) = \emptyset$  para todo vértice desse grafo, o que confirma ser  $g, f, c, e, d, b, a$  um esquema de eliminação perfeita.

Para verificar a correção do processo acima, observa-se que se  $L(v_j) - A(v_j) \neq \emptyset$ , então necessariamente existem vértices  $v_p, v_q$  com  $p < j < q$ , tais que (i)  $v_j, v_q \in A'(v_p)$ , sendo  $v_j$  o vértice de  $A'(v_p)$  mais próximo de  $v_p$  em  $S$ , e (ii)  $(v_j, v_q) \notin E$ . O vértice  $v_q$  satisfaz  $v_q \in L(v_j) - A(v_j)$ . Neste caso (Figura 4.19(a)),  $v_p$  não é simplicial. Logo,  $S$  não é um esquema de eliminação perfeita. Reciprocamente, seja a suposição de que

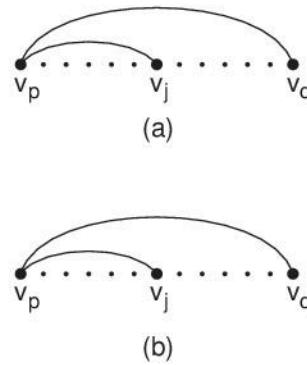


Figura 4.19: Correção: esquemas de eliminação perfeita

$L(v_j) - A(v_j) = \emptyset$ , para todo  $j$ ,  $1 \leq j \leq n$  e  $S$  não é um esquema de eliminação perfeita. Seja  $v_i$  o vértice não simplicial mais à direita de  $S$ , e denotemos por  $v_r \in A'(v_i)$  o vértice de  $A'(v_i)$  mais próximo de  $v_i$  em  $S$ . Então existe um vértice  $v_s \in A(v_i)$  tal que  $(v_r, v_s) \notin E$  (Figura 4.19(b)). Mas  $v_s \in L(v_r)$ , e  $L(v_r) - A(v_r) = \emptyset$  implica  $v_s \in A(v_r)$ , o que contradiz  $(v_r, v_s) \notin E$ .

Através da aplicação de técnicas de ordenação, o algoritmo descrito para reconhecimento de esquemas de eliminação perfeita pode ser implementado em tempo  $O(n + m)$ , o que garante a linearidade de todo o processo de reconhecimento de grafos cordais.

O algoritmo para efetuar o reconhecimento é, desta forma, simples. Seja  $G$  o grafo dado, o qual se deseja verificar se é ou não cordal. Executa-se uma busca em largura lexicográfica e obtém-se uma sequência  $S$  dos vértices de  $G$ , ordenados decrescentemente em suas larguras. Aplica-se então a  $S$  e  $G$  o algoritmo de reconhecimento de esquemas de eliminação perfeita. Foi visto que  $G$  é cordal se e somente se  $S$  for reconhecido como um tal esquema.

## 4.10 Busca Irrestrita

As propriedades e aplicações desenvolvidas no presente capítulo, até este ponto, referem-se à técnica de busca, tal qual foi conceituada na Seção 4.2. Uma decorrência dessa conceituação é que durante o processo cada aresta é visitada apenas um número constante de vezes (duas, por exemplo, na busca em profundidade em grafos não direcionados). Utiliza-se então o termo busca restrita para identificar um procedimento com esta característica.

Em contrapartida, uma *busca irrestrita* em um grafo  $G$  é um processo sistemático de se percorrer  $G$  de tal modo que cada aresta seja visitada um número finito (qualquer) de vezes. Note que segundo a definição, a única restrição ao processo é que ele deve terminar.

A ideia de busca em profundidade pode ser aplicada também para busca irrestrita. Considere o Algoritmo 4.2, de busca (restrita) em profundidade. Seja a seguinte alteração efetuada em sua estratégia: “durante o processo, um vértice  $v$  estará marcado se e somente se  $v$  pertencer à pilha  $Q$ ”. Isto é, cada vértice é marcado ao ser incluído em  $Q$  e desmarcado por ocasião da sua exclusão de  $Q$ . Nesse caso, obviamente, um vértice pode ser explorado diversas vezes durante a busca. De fato, suponha uma aplicação da busca com a alteração anteriormente incorporada. Em um certo instante, seja  $a$  o vértice que se encontra no topo da pilha  $Q$ . Isto é, seja a computação  $P(a)$ . Para  $b = w \in A(a)$ , a exploração da aresta  $(a, b)$  implicará o início imediato de exploração do vértice  $b$  (isto é, a chamada recursiva  $P(b)$ ) quando e somente quando o vértice  $b$  estiver fora da pilha  $Q$ . O algoritmo seguinte descreve o método.

Em relação ao algoritmo acima, seria razoável indagar se o mesmo termina (ou caso contrário, apresenta ciclicidade). Para concluir que o algoritmo termina de fato, observe que se um vértice  $v$  for colocado na pilha  $Q$  por mais de uma vez, então necessariamente a configuração em  $Q$ , abaixo de  $v$ , é diferente em cada caso. Como existe um número finito de modos de se arranjar essas configurações, o algoritmo necessariamente chega ao final. Ressalte-se que esta propriedade depende fortemente do fato de que qualquer vértice não pode ser reexplorado,

**Algoritmo 4.8:** Busca irrestrita em profundidade (utilizando pilha)**Dados:**  $G(V, E)$  conexo**Procedimento**  $P(v)$ 

```

    marcar  $v$ 
    colocar  $v$  na pilha  $Q$ 
    para  $w \in A(v)$  efetuar
        se  $w \notin Q$  então
            visitar  $(v, w)$ 
             $P(w)$ 
        retirar  $v$  de  $Q$ 
        desmarcar  $v$ 
    desmarcar todos os vértices
    definir uma pilha  $Q$ 
    escolher uma raiz  $s$ 
     $P(s)$ 

```

enquanto permanecer na pilha. A ocorrência de chamada recursiva  $P(w)$ , com  $w$  pertencendo à pilha  $Q$ , pode ocasionar ciclicidade (Figura 4.20).

O lema seguinte explica o funcionamento da pilha  $Q$  e, consequentemente, descreve o processo de busca irrestrita em profundidade.

**Lema 4.9**

Seja  $G(V, E)$  um grafo não direcionado conexo, aplicado ao Algoritmo 4.8. Seja  $v \in V$  incluído na pilha  $i$  vezes, durante o processo. Denote por  $C_i$  a configuração da pilha  $Q$ , abaixo de  $v$ , no momento em que o vértice  $v$  for incluído em  $Q$  pela  $i$ -ésima vez. Então necessariamente:

- (i) A configuração de  $Q$ , no momento em que  $v$  for excluído pela  $i$ -ésima vez da pilha, é também igual a  $C_i$ .
- (ii)  $i \neq j \Rightarrow C_i \neq C_j$ .

Uma busca irrestrita em profundidade, realizada num grafo  $G$ , constrói uma árvore  $T$ , enraizada e rotulada, denominada *árvore irrestrita de profundidade*, com as seguintes propriedades:

- (i) O rótulo da raiz de  $T$  é o vértice raiz da busca.
- (ii) Cada chamada recursiva  $P(w)$ , dentro do procedimento  $P(v)$  incluído no Algoritmo 4.8, corresponde em  $T$  a uma aresta  $(v', w')$ , sendo  $v'$  o pai de  $w'$ , com rótulo  $(v') = v$  e rótulo  $(w') = w$ .

Observe que a árvore irrestrita de profundidade traduz o funcionamento da pilha  $Q$  do Algoritmo 4.8. Por exemplo, a árvore da Figura 4.21(b) é a árvore irrestrita de profundidade em uma busca de raiz  $a$ , realizada no grafo da Figura 4.21(a), supondo-se que suas listas de adjacências estejam em ordem alfabética.

Pelo lema seguinte, conclui-se que a árvore irrestrita de profundidade de um grafo  $G$  independe da busca que a gerou, desde que considerada como não ordenada e de raiz fixa.

**Lema 4.10**

Uma árvore rotulada  $T$ , de raiz  $s$ , é árvore irrestrita de profundidade de algum grafo  $G$  se e somente se cada caminho maximal em  $G$  com um extremo em  $s$  corresponder univocamente a cada caminho em  $T$ , de uma folha até a raiz.

Uma consequência do Lema 4.10 é que o Algoritmo 4.8, na realidade, determina todos os caminhos maximais de  $G$  iniciados por  $s$ .

Há aplicações em que não se necessita gerar todos os caminhos maximais, mas apenas um certo número deles. Pode-se então restringir parcialmente a busca, de modo a admitir em determinadas condições vértices marcados fora da pilha  $Q$  (o que, naturalmente, faz diminuir o número de explorações de vértices). Isto é, todo vértice  $v$  é

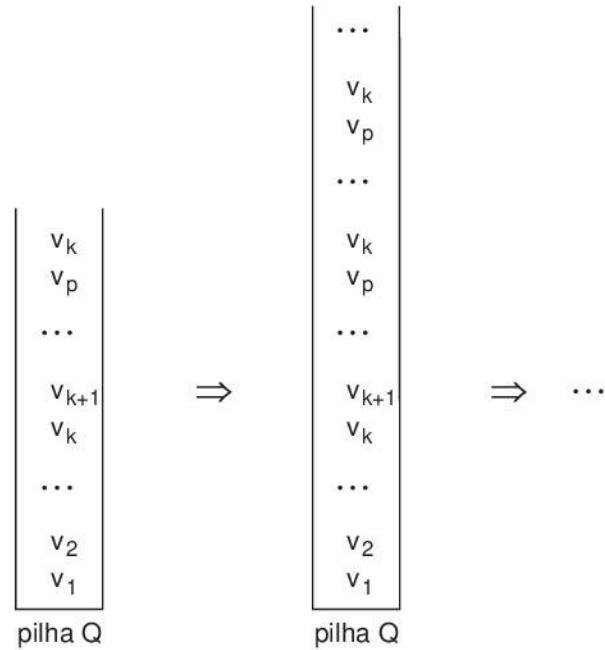


Figura 4.20: Ciclicidade

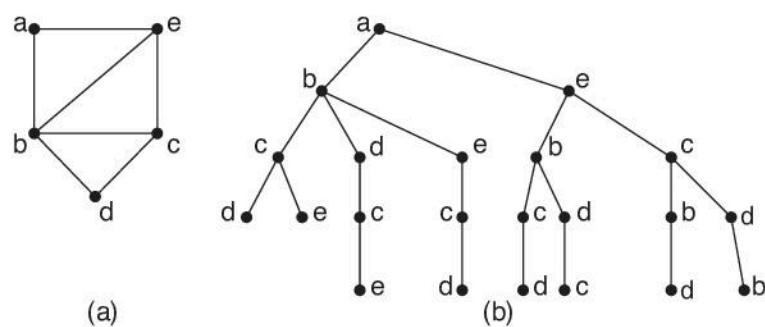


Figura 4.21: Busca irrestrita em profundidade de raiz  $a$

marcado quando é incluído na pilha  $Q$ . Ao ser excluído de  $Q$ , o vértice  $v$  pode ser desmarcado ou não, dependendo de condições específicas à aplicação em questão. Assim sendo, se  $v \in Q$  e  $v$  for alcançado na busca, então  $v$  não será explorado nessa ocasião. Se  $v \notin Q$  e for alcançado na busca, o vértice poderá ou não ser explorado. Como exemplo, um problema que admite algoritmos eficientes baseados nessa técnica é o de se enumerar todos os ciclos simples de um grafo.

Analogamente ao Algoritmo 4.2, de busca em profundidade recursiva em grafos não direcionados, o Algoritmo 4.8 de busca irrestrita em profundidade, admite uma versão puramente recursiva, sem a utilização explícita da pilha  $Q$ . Esta versão é descrita no Algoritmo 4.9.

---

**Algoritmo 4.9:** Busca irrestrita em profundidade (recursivo)

---

**Dados:**  $G(V, E)$  conexo

**Procedimento**  $P(u, v)$

    marcar  $v$

    para  $w \in A(v)$  efetuar

        se  $w$  é não marcado então

            visitar aresta de árvore  $(v, w)$

$P(w, v)$

        caso contrário

            se  $w \neq u$  então

                visitar aresta de retorno  $(v, w)$

    desmarcar  $v$

    desmarcar todos os vértices

    escolher uma raiz  $s$

$P(s, \emptyset)$

---

## 4.11 Programas em Python

Esta seção contém implementações dos algoritmos formulados neste capítulo. As implementações seguem as descrições gerais apresentadas nos Capítulos 1 e 2.

### 4.11.1 Algoritmo 4.2: Busca em Profundidade

O Programa 4.1 corresponde à implementação do Algoritmo 4.2. Nesta implementação, a marcação de um vértice (Linhas 5 e 17) está representada por um vetor `Marcado` de valores lógicos para acesso direto. Desta forma, procedimentos que invocam a busca em profundidade podem também ter acesso a tal marcação externamente para certos propósitos. Como exemplo, a busca em profundidade poderia ser utilizada para grafos gerais e o vetor `Marcado` seria útil para se verificar se o grafo é conexo (um grafo é conexo se e somente se todos os vértices encontram-se marcados após a busca). O vetor `EmQ` é outro vetor de valores lógicos para acesso direto que representa se um dado vértice está em  $Q$ . A condição da Linha 12 justifica-se da seguinte forma: como  $v$  encontra-se no topo da pilha  $Q$ ,  $v, w$  não serem consecutivos em  $Q$  equivale a verificar se  $w$  não é o penúltimo elemento de  $Q$ . O restante do algoritmo é tradução direta para a linguagem.

#### Programa 4.1: Busca em profundidade (utilizando pilha)

```

1 #Algoritmo 4.2: Busca em profundidade (utilizando pilha)
2 #Dado: grafo conexo G
3 def BuscaProfundidade(G):
4     def P(v):
5         G.Marcado[v], G.EmQ[v] = True, True
6         Q.append(v)
7         for w in G.N(v):

```

```

8     if not G.Marcado[w]:
9         Visitar(v, w)      #arestas visitadas (I)
10        P(w)
11    else:
12        if G.EmQ[w] and w != Q[-2]:
13            Visitar(v, w)  #arestas visitadas (II)
14        G.EmQ[v] = False
15        Q.pop()
16
17 G.Marcado, G.EmQ = [False]*(G.n+1), [False]*(G.n+1)
18 Q = []
19 s = 1
20 P(s)

```

#### 4.11.2 Algoritmo 4.4: Busca em Profundidade em Digrafos

O Programa 4.2 corresponde à implementação direta do Algoritmo 4.4. A marcação dos vértices é feita tal como no Algoritmo 4.2.

Programa 4.2: Busca em profundidade em digrafos

```

1 #Algoritmo 4.4: Busca em profundidade em digrafos
2 #Dado: digrafo D
3 def BuscaProfundidade(D):
4     def P(v):
5         D.Marcado[v] = True
6         Q.append(v)
7         for w in D.N(v, "+"):
8             Visitar(v, w)
9             if not D.Marcado[w]:
10                P(w)
11        Q.pop()
12
13 D.Marcado = [False]*(D.n+1)
14 Q = []
15 s = 1
16 P(s)

```

#### 4.11.3 Algoritmo 4.5: Componentes Fortemente Conexas

O Programa 4.3 corresponde à implementação direta do Algoritmo 4.5. A marcação dos vértices é feita tal como no Algoritmo 4.2.

Programa 4.3: Componentes fortemente conexas de um digrafo

```

1 #Algoritmo 4.5: Componentes fortemente conexas de um digrafo
2 #Dado: digrafo D
3 def CompFortementeConexas(D):
4     def F(v):
5         global j
6         D.Marcado[v] = True
7         j = j + 1
8         PE[v], low[v] = j, j
9         for w in D.N(v,Tipo="+"):
10            if not D.Marcado[w]:

```

```

11         T.AdicionarAresta(v,w)
12         F(w)
13         low[v] = min(low[v], low[w])
14     else:
15         if T.L[w] != None:
16             low[v] = min(low[v], PE[w])
17     if low[v] == PE[v]:
18         #retirar v e seus descendentes de T
19     global j
20     j = 0
21     D.Marcado, PE, low = [False]*(D.n+1), [0]*(D.n+1), [0]*(D.n+1)
22
23     T = GrafoListaAdj(orientado=False)
24     T.DefinirN(D.n)
25     for s in D.V():
26         if not D.Marcado[s]:
27             F(s)

```

#### 4.11.4 Algoritmo 4.6: Busca em Largura

O Programa 4.4 corresponde à implementação direta do Algoritmo 4.6. A marcação dos vértices é feita tal como no Algoritmo 4.2.

Programa 4.4: Busca em largura

```

#Algoritmo 4.6: Busca em largura
#Dado: Grafo conexo G
def BuscaLargura(G):
    G.Marcado, G.EmQ = [False]*(G.n+1), [False]*(G.n+1)
    s = 1
    Q = deque()
    G.Marcado[s], G.EmQ[s] = True, True
    Q.append(s)
    while len(Q) > 0:
        v = Q[0]
        for w in G.N(v):
            if not G.Marcado[w]:
                Visitar(v, w)      #arestas visitadas (I)
                G.Marcado[w], G.EmQ[w] = True, True
                Q.append(w)
            else:
                if G.EmQ[w]:
                    Visitar(v, w) #arestas visitadas (II)
                    G.EmQ[v] = False
                    v = Q.popleft()

```

#### 4.11.5 Algoritmo 4.7: Busca em Largura Lexicográfica

A implementação do Algoritmo 4.7 é diretamente traduzida para a linguagem. Observe que a obtenção do rótulo lexicograficamente máximo é delegada à função EncontrarMax. Note que, nesta função, os elementos sendo comparados em  $\text{maxR} < R[v]$  são listas e, em Python, uma lista é menor do que outra justamente quando a primeira é lexicograficamente menor que a última.

Programa 4.5: Busca em largura lexicográfica

```

1 #Algoritmo 4.7: Busca em largura lexicográfica
2 #Dado: Grafo conexo G
3 def BuscaLarguraLexico(G):
4     G.Marcado = [False]*(G.n+1)
5     R = [None]*(G.n+1)
6     for v in range(1, G.n+1):
7         R[v] = [0]
8     Q = []
9     for j in range(G.n, 0, -1):
10        v = EncontrarMax(R)
11        G.Marcado[v], R[v] = True, []
12        Q.append(v)
13        for w in G.N(v):
14            if not G.Marcado[w]:
15                R[w].append(j)
16    return Q
17
18 def EncontrarMax(R):
19     maxR = None
20     vmax = None
21     for v in range(1, len(R)):
22         (maxR, vmax) = (R[v], v) if (maxR==None or maxR<R[v]) else (maxR, vmax)
23     return vmax

```

#### 4.11.6 Algoritmo 4.8: Busca Irrestrita

O Programa 4.6 corresponde à implementação direta do Algoritmo 4.8. A marcação dos vértices é feita tal como no Algoritmo 4.2.

Programa 4.6: Busca irrestrita em profundidade (utilizando pilha)

```

1 #Algoritmo 4.8: Busca irrestrita em profundidade (utilizando pilha)
2 #Dado: Grafo conexo G
3 def BuscaIrrestrita(G):
4     def P(v):
5         G.Marcado[v] = True
6         Q.append(v)
7         for w in G.N(v):
8             if not G.Marcado[w]:
9                 Visitar(G, v, w)
10                P(w)
11            Q.pop()
12            G.Marcado[v] = False
13
14    G.Marcado = [False]*(G.n+1)
15    Q = []
16    s = 1
17    P(s)

```

### 4.12 Exercícios

- Efetuar uma busca geral, segundo o Algoritmo 4.1, no grafo da Figura 4.2.

- 4.2 Qual o efeito da aplicação do Algoritmo 4.1 em um grafo desconexo?
  - 4.3 Formular uma descrição alternativa do Algoritmo 4.2, de busca em profundidade, de modo a eliminar a pilha  $Q$ .
  - 4.4 Seja  $G(V, E)$  um grafo e  $(v, w) \in E$ . Formular um processo para reconhecer se  $(v, w)$  é aresta de árvore ou retorno em relação a uma busca em profundidade, a partir das profundidades de entrada e saída dos vértices de  $G$ .
  - 4.5 Um grafo  $G$  é uma árvore se e somente se uma busca em profundidade efetuada em  $G$  não produzir arestas de retorno. Provar ou dar contra-exemplo.
  - 4.6 Seja  $G(V, E)$  um grafo e  $A(V, E_A)$  uma árvore geradora de  $G$ . Elaborar um algoritmo para reconhecer se  $A$  é árvore geradora de profundidade de  $G$ .
  - 4.7 Provar que num processo completo do Algoritmo 4.2, de busca em profundidade, cada aresta do grafo  $G$  é examinada exatamente duas vezes.
  - 4.8 Que ordenação das listas de adjacências do grafo da Figura 4.4(a) produz a busca da Figura 4.4(b)?
  - 4.9 Seja  $B$  uma busca em profundidade efetuada num grafo não direcionado  $G(V, E)$ . Em que condições vale:  $\text{PE}(v) < \text{PE}(w) \Rightarrow \text{PS}(v) < \text{PS}(w)$ , para todo  $(v, w) \in V$ ?
  - 4.10 Seja  $G$  um grafo não direcionado e  $v$  uma articulação de  $G$ . O número de componentes biconexas de  $G$  que contêm  $v$  é igual ao número de filhos de  $v$ , em uma árvore de profundidade de  $G$ . Provar ou dar contra-exemplo.
  - 4.11 O número de componentes biconexas de um grafo  $G$  é igual ao número de demarcadores, obtidos a partir de uma busca em profundidade arbitrária. Provar ou dar contra-exemplo.
  - 4.12 Caracterizar os grafos para os quais todo filho de articulação é demarcador.
  - 4.13 Escrever um algoritmo para, explicitamente, determinar as componentes biconexas de um grafo não direcionado além dos pontos de articulação. A complexidade do algoritmo deve ser linear no número de vértices e arestas de  $G$ .
- Sugestão. Ver Seção 4.4.
- 4.14 Modificar o algoritmo de biconectividade do exercício anterior de modo a encontrar todas as pontes do grafo.
  - 4.15 Formular uma descrição do Algoritmo 4.4, de busca em profundidade em digrafos, de modo a eliminar a pilha  $Q$ .

- 4.16 Seja  $D(V, E)$  um digrafo,  $(v, w) \in E$ , e  $B$  uma busca em profundidade no digrafo. Provar que se  $(v, w)$  é aresta de avanço então  $\text{PE}(v) < \text{PE}(w) + 1$ . Vale a recíproca?
- 4.17 Estender o Algoritmo 4.4, que efetua uma busca em profundidade em digrafos, para obter também uma partição das arestas em arestas de árvore, avanço, cruzamento e retorno, respectivamente. A complexidade deve ser mantida.
- 4.18 Descrever um algoritmo baseado em busca em profundidade, para realizar uma ordenação topológica de um grafo direcionado acíclico  $D$ . A complexidade do algoritmo deve ser linear no número de vértices e arestas de  $D$ .
- 4.19 Descrever um algoritmo baseado em busca em profundidade, para determinar o caminho de comprimento máximo em um grafo direcionado acíclico  $D$ . A complexidade do algoritmo deve ser linear no número de vértices e arestas de  $D$ .
- 4.20 Seja  $D$  um digrafo dado. Elaborar um algoritmo para reconhecer se uma dada árvore direcionada enraizada  $T$  é árvore de profundidade de  $D$ .
- 4.21 Um digrafo  $D$  é acíclico se e somente se existir uma busca em profundidade em  $D$  sem arestas de retorno. Provar ou dar contra-exemplo.
- 4.22 Caracterizar os digrafos  $D(V, E)$  com raiz  $s \in V$ , para os quais toda busca em profundidade com raiz  $s$  não produz arestas de cruzamento.
- 4.23 Seja  $D(V, E)$  um digrafo. Uma aresta  $(v, w)$  é *implícita por transitividade* quando existir um caminho de  $v$  para  $w$  em  $D$  que não contém  $(v, w)$ . Então uma aresta  $(v, w)$  é implícita por transitividade se e somente se  $(v, w)$  é aresta de avanço em qualquer busca em profundidade em  $D$ . Provar ou dar contra-exemplo.
- 4.24 Seja  $D$  um digrafo acíclico, em que cada lista de adjacências encontra-se em uma ordenação topológica reversa. Então toda busca em profundidade efetuada em  $D$  não produz arestas de avanço. Provar ou dar contra-exemplo.
- 4.25 Seja  $D(V, E)$  um digrafo e  $(v, w) \in E$  tal que (i)  $v, w$  pertencem a componentes fortemente conexas distintas de  $D$  e (ii)  $\text{PE}(w) < \text{PE}(v)$  em uma busca em profundidade efetuada em  $D$ . Então no Algoritmo 4.4,  $w \notin Q$  no momento da visita à aresta  $(v, w)$ . Provar ou dar contra-exemplo.
- 4.26 Seja a computação da função  $\text{low}$ , conforme descrita na Seção 4.6. Provar que a mesma está correta.
- 4.27 Descrever outra formulação do Algoritmo 4.5, empregando uma pilha em lugar de árvore  $T$ .

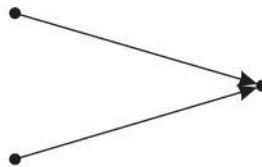


Figura 4.22: O subgrafo proibido

- 4.28 Um digrafo é *estruturado em árvore* quando for acíclico e tal que sua redução transitiva seja uma árvore. Provar que se  $D$  for estruturado em árvore existe uma busca em profundidade em  $D$  que não produz arestas de cruzamento.
- 4.29 Provar que um digrafo  $D$  é estruturado em árvore se e somente se seu fecho transitivo não contiver o digrafo da Figura 4.22 como subgrafo induzido.
- 4.30 Seja uma busca em profundidade realizada num grafo não direcionado e conexo  $G$ . Seja  $D$  o digrafo obtido orientando-se cada aresta  $(v, w)$  de  $G$  da menor para a maior profundidade de entrada de seus vértices. Provar que  $D$  é estruturado em árvore.
- 4.31 Descrever um algoritmo de complexidade linear para reconhecer digrafos estruturados em árvore.
- 4.32 Descrever um algoritmo de complexidade linear para determinar a redução transitiva de um digrafo estruturado em árvore.
- 4.33 Descrever um algoritmo para determinar o fecho transitivo de um digrafo estruturado em árvore, tal que possua complexidade linear no tamanho de sua saída.
- 4.34 Um digrafo *série paralelo minimal* (SPM) é definido recursivamente por:
- um digrafo trivial (com um único vértice) é SPM e
  - se  $D_1(V_1, E_1)$  e  $D_2(V_2, E_2)$  são digrafos SPM disjuntos então também o são os digrafos  $(V_1 \cup V_2, E_1 \cup E_2)$  e  $(V_1 \cup V_2, E_1 \cup E_2 \cup (B_1 \times A_2))$ , onde  $B_1, A_2$  são os conjuntos dos sumidouros de  $D_1$  e fontes de  $D_2$ , respectivamente.
- Dê exemplo de um digrafo que seja estruturado em árvore, mas não SPM. Em seguida, exemplifique um outro que seja SPM, mas não estruturado em árvore.
- 4.35 Um digrafo acíclico é *série paralelo geral* (SPG) quando a sua redução transitiva for SPM. Mostrar que todo digrafo estruturado em árvore é SPG.
- 4.36 Elaborar um algoritmo para reconhecer se num dado digrafo acíclico  $D$ , com raiz  $r$ , pode ser efetuada uma busca em profundidade, com raiz  $r$  e que produza no máximo  $k$  arestas de cruzamento, para um certo inteiro  $k$  (a que corresponde o caso  $k = 0$ ?).

- 4.37 Mostrar que as arestas de todo grafo não direcionado  $G$  podem ser orientadas de tal forma que o digrafo resultante seja acíclico. Apresentar o algoritmo correspondente, de complexidade linear.
- 4.38 É possível orientar as arestas de um grafo não direcionado  $G$ , de modo a obter um digrafo acíclico e sem arestas implícitas por transitividade? E se  $G$  não contiver triângulos?
- 4.39 Mostrar que no Algoritmo 4.6 de busca em largura, cada aresta do grafo é visitada exatamente duas vezes.
- 4.40 Elaborar um algoritmo para reconhecer se uma dada árvore é ou não árvore de largura de um dado grafo.
- 4.41 Provar que um grafo  $G$  é bipartido se e somente se uma busca em largura em  $G$  não produzir arestas primo nem irmão.
- 4.42 Desenvolver um algoritmo para efetuar uma busca em largura em digrafos. Quantos tipos de arestas diferentes produz essa busca?
- 4.43 Utilizando busca em largura, descrever um algoritmo para obter uma árvore geradora enraizada de um grafo  $G$ , com altura mínima. Qual a complexidade do processo?
- 4.44 Numa busca, a ordem de exploração dos vértices corresponde à ordem em que os mesmos são incluídos em  $Q$ , nos seguintes casos:
- a busca é de profundidade, e  $Q$  é a pilha associada à busca.
  - a busca é de largura, e  $Q$  é a fila associada à busca.
- Provar ou dar contra-exemplo.
- 4.45 É possível definir a seguinte variação para busca em largura lexicográfica. Ao invés de utilizar um critério de escolha de arestas que selecione a aresta mais remota (como a do Algoritmo 4.7), escolher sempre a aresta menos remota. Provar ou dar contra-exemplo.
- 4.46 Definir o conceito de busca em profundidade lexicográfica, análogo ao da busca em largura lexicográfica (sugestão: estabelecer condições para que um vértice  $v$  seja mais remoto do que um vértice  $w$ , e definir um critério de escolha de arestas baseado nessas condições e aplicável à busca em profundidade). Qual a complexidade desse processo?
- 4.47 Caracterizar os grafos  $G(V, E)$  para os quais o número de modos diferentes de efetuar busca em largura lexicográfica em  $G$  é polinomial em  $|V|$ .
- 4.48 Mostrar que a condição “escolher um vértice  $v$  marcado com  $R(v)$  lexicograficamente máximo”, no Algoritmo 4.7 de busca em largura lexicográfica, satisfaz à escolha de  $v$  através da utilização de uma fila.

- 4.49 Modificar o Algoritmo 4.7 de busca em largura lexicográfica, de modo a obter explicitamente a árvore de largura correspondente.
- 4.50 Determinar o grafo  $G$  tal que durante qualquer processo de busca em largura lexicográfica em  $G$ , a escolha de aresta incidente é sempre única, à exceção dos filhos da raiz. O grafo  $G$  deve satisfazer ainda às seguintes condições:

- (i) A raiz da busca é um vértice fixo de grau 2.
- (ii) O número de arestas de  $G$  é máximo.
- (iii) A árvore de largura possui altura 2.

Repetir o problema supondo que a árvore de largura possua altura 3, ao invés de 2. Repetir novamente, com a condição de que o grau da raiz fixa seja igual a 3, ao invés de 2.

- 4.51 Numa busca em largura lexicográfica, definir um critério absoluto de escolha de aresta  $(v, w)$  incidente ao vértice marcado  $v$ , quando  $w \in A(v) - A^*(v)$ , isto é,  $w \in A(v)$  se encontra marcado. Estender esse critério para a busca geral.
- 4.52 Reconhecer se uma árvore dada é ou não árvore de largura lexicográfica de um grafo não direcionado dado.
- 4.53 Formular uma implementação detalhada do algoritmo de reconhecimento de grafos cordais, descrito na Seção 4.9.
- 4.54 Um grafo é *4-cordal* quando todo ciclo de comprimento  $> 4$  possui uma corda. Elaborar um algoritmo para reconhecer grafos 4-cordais.
- 4.55 Seja  $T$  uma árvore irrestrita de profundidade de um grafo  $G$ . Provar que os rótulos de  $T$  (vértices de  $G$ ), correspondentes a um caminho em  $T$  da raiz até uma folha, são todos diferentes entre si.
- 4.56 Descrever um algoritmo para busca irrestrita em largura.

- 4.57 Seja  $B$  uma busca em profundidade, com raiz  $r$ , em um grafo  $G$  e  $Q$  a pilha associada à busca. Então o tamanho máximo que  $Q$  atinge durante o processo de busca é igual ao maior caminho em  $G$ , com extremidade em  $r$ , nos seguintes casos:
- (i)  $B$  é uma busca restrita em profundidade.
  - (ii)  $B$  é uma busca irrestrita em profundidade.

Provar ou dar contra-exemplo.

- 4.58 Durante um processo de busca restrita efetuada num grafo  $G$ , considere o conjunto  $S = \{v_i\}$ , tal que  $v_i$  é um vértice marcado e incidente a (pelo menos) uma aresta ainda não explorada. Os vértices de  $S$  correspondem aos candidatos para a próxima escolha de vértice marcado, na busca. Suponha o seguinte critério de escolha para tal vértice:

“dentre todos os vértices marcados e incidentes a uma aresta ainda não explorada, escolher aquele que é o  $\lfloor |S|/2 \rfloor$  mais recentemente alcançado na busca”.

Elaborar um algoritmo para efetuar uma busca em um grafo utilizando o critério acima. Aplicar o algoritmo ao grafo  $K_6$ , com conjunto dos vértices  $\{1, 2, 3, 4, 5, 6\}$  e cujas listas de adjacência estão ordenadas em ordem crescente de rótulos dos respectivos vértices.

4.59 Provar o Lema 4.9.

4.60 O seguinte algoritmo se destina a enumerar os ciclos simples de um digrafo.

**Algoritmo:** enumeração de ciclos

**Dados:** digrafo  $D(V, E)$

**Procedimento**  $C(v)$

inserir  $v$  em  $Q$

para  $w \in A(v)$  efetuar

se  $w \notin Q$  então

$C(w)$

**caso contrário**

seja  $w, \dots, v$  a sequência de vértices em  $Q$ , de  $w$  para o topo

listar ciclo  $w, \dots, v, w$

retirar  $v$  de  $Q$

definir pilha  $Q$

escolher  $s \in V$

$C(s)$

O algoritmo acima apresenta incorreções. Pede-se corrigi-las.

4.61 POSCOMP-2014

Considere as afirmativas a seguir, sobre grafos.

- I A busca em profundidade em um grafo não direcionado irá produzir arestas de árvore e de cruzamento.
- II A busca em profundidade decompõe um grafo direcionado em suas componentes fortemente conexas.
- III Um grafo direcionado é acíclico quando uma busca em profundidade não produzir arestas de retorno.
- IV Uma ordenação topológica de um grafo é uma ordenação linear de seus vértices.

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.

4.62 UVA Online Judge 1220

Escreva um algoritmo para o seguinte problema:

Um empregado da empresa BCM vai dar uma festa e quer convidar os colegas de trabalho. Ele não vai convidar simultaneamente nenhum empregado e seu chefe imediato. Dada a descrição hierárquica da empresa, determinar o máximo de pessoas que serão convidadas. Indicar, também, se a solução é única.

#### 4.63 UVA Online Judge 12160

Escreva um algoritmo para o seguinte problema:

O segredo de uma fechadura é um inteiro  $n$  de 4 dígitos. A fechadura tem  $m$  botões, cada um com um inteiro associado. Esses inteiros são conhecidos. Existe um visor com um valor inicial  $u$ . Cada vez que um dos  $m$  botões é acionado o valor é somado (em módulo 10000) ao visor e o resultado passa a ser o valor corrente no visor. Determine se é possível abrir o cadeado e qual o número mínimo de acionamentos do botão para conseguir a abertura.

#### 4.64 UVA Online Judge 610

Escreva um algoritmo para o seguinte problema:

O trânsito em uma cidade é modelado através de um grafo, onde os vértices são as esquinas e as arestas, os trechos de rua entre duas esquinas. Hoje em dia o trânsito é de mão dupla em todos os trechos de rua e é possível alcançar cada esquina a partir de qualquer outra. Os órgãos de trânsito consideram que há muitos acidentes causados pelas mãos duplas e o prefeito quer reorientar o trânsito tal que o maior trecho de ruas fique com mão única. Indicar uma possível solução para isso, imprimindo, para os trechos de rua que podem ficar com mão única, o par  $(ij)$ , com o trânsito fluindo da esquina  $i$  para a esquina  $j$ . Para os trechos que devam permanecer com mão dupla, imprima os pares  $(i, j)$  e  $(j, i)$ .

#### 4.65 UVA Online Judge 11504

Faça um programa para o seguinte problema:

Dominós são divertidos. Crianças gostam de colocar em pé as peças em longas filas. Quando um dominó cai, ele derruba o próximo e assim por diante. Entretanto, algumas vezes não se consegue derrubar todo o conjunto começando apenas por uma peça. Sua tarefa é determinar, dado o layout de um conjunto de peças, o número mínimo de dominós que devem ser derrubados para, em série, derrubar-se todo o conjunto.

### 4.13 Notas Bibliográficas

As técnicas de busca, inclusive em profundidade, já são conhecidas desde o século XIX, pelo menos. Um algoritmo semelhante ao 4.2, de autoria de Trémaux, foi comunicado por Lucas (1882). Pouco mais tarde foi publicado o algoritmo de Tarry (1895). Veja artigo de Fraysseix, Ossona de Mendez e Rosenstieh (2006), a respeito. Naquela ocasião, esses algoritmos eram utilizados na solução de problemas de caminhamento em labirintos. Mais recentemente, a técnica voltou a ser empregada na solução de problemas combinatórios, com o nome de “backtracking” (Golomb e Baumert (1965), Floyd (1967), Knuth (1975)). Contudo, a utilização extensiva da busca, como técnica eficaz para resolver diversos problemas algorítmicos em grafos, foi iniciada por Tarjan, no princípio da década de 1970. Em Tarjan (1972), o algoritmo de busca em profundidade é apresentado em detalhe, sendo descritas suas propriedades básicas, correspondentes aos Teoremas 4.1 e 4.2. Nesse mesmo artigo são apresentadas as aplicações de determinação das componentes biconexas de um grafo (Seção 4.4) e das componentes fortemente conexas de um digrafo (Seção 4.6). Veja também o livro de Tarjan (1983). O Teorema 4.3 aparece em Szwarcfiter, Persiano e Oliveira (1985). Vários textos descrevem algoritmos de busca em grafos. O livro de Cormen, Leiserson, Rivest e Stein (2001) é bastante conhecido. Dentre as inúmeras aplicações de busca que se seguiram ao mencionado artigo de Tarjan, as seguintes podem ser mencionadas: determinação das componentes triconexas de um grafo (Hopcroft e Tarjan (1973)),

reconhecimento de grafos planares (Hopcroft e Tarjan (1974)), enumeração dos ciclos simples de um digrafo (Tarjan (1973), Johnson (1975), entre outros). Um trabalho sobre aplicações de busca, em língua portuguesa, é Méxas (1982). Read e Tarjan (1975) apresentaram algoritmos para enumerar as árvores geradoras de um grafo e os ciclos simples de um digrafo. A determinação de dominadores em um digrafo pode ser encontrada em Tarjan (1974) e o reconhecimento de digrafos redutíveis em Tarjan (1974a). Uma outra referência relevante desta série é Hopcroft e Tarjan (1973a), a qual contém implementações detalhadas de alguns algoritmos de busca em profundidade. A lista de aplicações é bastante vasta, incluindo-se diversos algoritmos elaborados também recentemente. Busca em largura lexicográfica foi descrita em Rose, Tarjan e Lueker (1976) e Sethi (1975). A busca em largura lexicográfica mostrou-se uma poderosa técnica para a utilização em reconhecimento de classes de grafos e outras aplicações para resolver problemas algorítmicos em grafos. Neste sentido, mencionamos os artigos de Figueiredo, Meidanis e Mello (1995) e o de Habib, McConnell, Paul e Viennot (2000), entre outros. Um tutorial abrangente sobre o assunto foi escrito por Corneil (2004). A ideia da implementação do algoritmo de reconhecimento de grafos cordais encontra-se em Lueker (1974) e Rose e Tarjan (1975). Os Teoremas 4.7 e 4.8 são de Dirac (1961) e 4.9 de Fulkerson e Gross (1965). Várias provas apresentadas na Seção 4.9 foram elaboradas a partir de Golumbic (1980). Os digrafos estruturados em árvore, Exercícios 4.28 e 4.35. podem ser encontrados em Szwarcfiter (1980). Os digrafos série paralelo, Exercícios 4.34 e 4.35, foram estudados inicialmente por Valdes (1978) e Valdes, Tarjan e Lawler (1979).

# OUTRAS TÉCNICAS

---

## 5.1 Introdução

O Capítulo 5 prossegue com a apresentação de técnicas gerais aplicáveis a algoritmos em grafos. No Capítulo 3 foram introduzidas as técnicas básicas, enquanto que no seguinte foi realizado um estudo de busca. Agora, são apresentadas três técnicas adicionais, a saber: *algoritmo guloso*, *programação dinâmica* e *condensação*. Como exemplos de aplicação dessas técnicas, respectivamente, são apresentados algoritmos para obter uma árvore geradora de peso mínimo de um grafo, partitionar uma árvore de maneira ótima e determinar o número cromático de um grafo.

Na realidade, as técnicas de algoritmo guloso e programação dinâmica são provenientes da área de otimização combinatória, a qual possui várias partes em comum com algoritmos em grafos. Essa interseção compreende, normalmente, problemas envolvendo grafos com valores numéricos (tais como pesos em arestas, vértices etc.). Em geral, o objetivo de um tal problema consiste em obter uma configuração desejada (por exemplo, algum caminho, uma partição de vértices etc.) de modo a otimizar um critério dado.

## 5.2 Algoritmo Guloso

A técnica do algoritmo guloso possui como característica importante a simplicidade. Isto é, uma solução de um problema, obtida através de sua aplicação, é quase sempre de natureza simples.

Considere o seguinte problema geral. Seja dado um conjunto  $S$ . Deseja-se determinar um subconjunto  $S' \subseteq S$  tal que:

- (i)  $S'$  satisfaz uma dada propriedade  $P$ , e
- (ii)  $S'$  é máximo (ou mínimo) em relação a algum critério dado  $\alpha$ .

Ou seja,  $S'$  é o maior (ou menor) subconjunto de  $S$ , segundo  $\alpha$ , que satisfaz  $P$ . O algoritmo guloso para resolver este problema consiste num processo iterativo em que  $S'$  é construído, adicionando-se ao mesmo elementos de  $S$ , um a um. Seja  $S'_{i-1}$  o subconjunto assim construído após a iteração  $i - 1$ . Na iteração  $i$  determina-se o elemento  $s \in S - S'_{i-1}$  tal que:

- (i)  $S'_{i-1} \cup \{s\}$  satisfaz  $P$ , e
- (ii)  $s$  é tal que  $S'_{i-1} \cup \{s\}$  é maior (menor) ou igual, segundo  $\alpha$ , do que  $S'_{i-1} \cup \{r\}$  que satisfaz  $P$ , para todo  $r \in S - S'_i$ .

Ou seja, a cada passo  $i$ , determina-se o elemento  $s \in S$  que adicionado a  $S'_{i-1}$  maximiza  $\alpha$ , (minimiza  $\alpha$ ) e além disso satisfaz  $P$ .

Observe que a denominação algoritmo guloso provém do fato de que a cada passo procura-se incorporar ao subconjunto até então construído a melhor porção possível, compatível com a propriedade  $P$ . Obviamente, há ocasiões em que a aplicação dessa estratégia não conduz ao resultado exato do problema. Ou seja, o processo garante que o subconjunto  $S'$  obtido satisfaz  $P$ , visto que este fato é verificado passo a passo, no algoritmo. Contudo, há aplicações em que não se pode garantir a maximalidade (minimalidade) de  $S'_i$ , obtido segundo o processo apresentado. Assim sendo, para a aplicação desse método, é necessária uma prova de que o subconjunto obtido seria de fato máximo (mínimo). A existência ou não dessa prova, naturalmente, depende do problema específico em consideração. Na Seção 5.3, examina-se um problema que pode ser resolvido por essa técnica.

Finalmente, observa-se que uma característica fundamental do guloso é que o subconjunto  $S'$ , iterativamente construído, jamais é alterado durante o processo. Exceto, obviamente, por novas adições. Ou seja, não se retiram elementos de  $S'$  durante a sua construção. Este fato é importante para a finalidade de determinar as complexidades dos algoritmos.

### 5.3 Árvore Geradora Mínima

Seja  $G(V, E)$  um grafo conexo, em que cada aresta  $e = (v, w)$  possui um peso  $d(e)$ . Denomina-se *peso* de uma árvore geradora  $T(V, E_T)$  de  $G$  à soma dos pesos das arestas de  $G$  que formam  $T$ , ou seja o peso de  $T$  é  $\sum_{e \in E_T} d(e)$ . O problema que se examina na presente seção é o de obter a árvore geradora de peso mínimo, para um dado grafo. A solução consiste em uma aplicação do algoritmo guloso.

Inicialmente, reformula-se o enunciado do problema nos termos da seção anterior. Isto é, deseja-se obter um subconjunto  $E_T \subseteq E$ , tal que:

- (i)  $(V, E_T)$  seja uma árvore, e
- (ii)  $\sum_{(v,w) \in E_T} d(v, w)$  seja mínimo.

Ou seja, a propriedade  $P$  mencionada na descrição geral do algoritmo (seção anterior) corresponde a escolher  $E_T$  de tal modo que  $(V, E_T)$  seja uma árvore geradora. O critério de otimização corresponde a minimizar a soma dos pesos das arestas de  $E_T$ .

O guloso correspondente é o seguinte. Definir, inicialmente,  $E_T = \emptyset$ . A cada passo, escolher uma aresta  $(v, w)$  ainda não considerada tal que (i) a incorporação de  $(v, w)$  ao conjunto de arestas  $E_T$  até então obtido não produz ciclos e (ii) o peso total de  $E_T \cup \{(v, w)\}$  é mínimo, dentre todas as escolhas de arestas que satisfazem (i). Após essa verificação, simplesmente incorporar  $(v, w)$  a  $E_T$  e repetir o processo, até que todas as arestas tenham sido consideradas.

A aresta  $(v, w)$  que minimiza o peso de  $E_T \cup \{(v, w)\}$  é evidentemente a aresta de menor peso ainda não considerada. Este fato simplifica a operação correspondente a (ii). Assim sendo, o algoritmo pode ser descrito, de forma simples, pela seguinte sentença:

“*incluir em  $E_T$   
todas as arestas de  $E$   
em ordem não decrescente de peso  
rejeitando  
cada uma que formar ciclo  
com aquelas já incluídas*”.

Observe que o algoritmo acima pode ser interpretado como sendo a construção de uma árvore geradora, a partir de uma floresta. O estado inicial corresponde ao de uma floresta formada por  $n$  árvores triviais (um só vértice, cada), o que significa  $E_T = \emptyset$ . Cada inclusão de aresta  $(v, w)$  ao conjunto  $E_T$  é interpretada como a fusão das duas árvores da floresta que contêm  $v$  e  $w$ , respectivamente, em uma única. Uma aresta  $(v, w)$  é rejeitada precisamente quando  $v, w$  pertencerem a uma mesma árvore. O processo se encerra quando todas as árvores da floresta tiverem sido fundidas em uma só, através da escolha de  $n - 1$  arestas.

As figuras seguintes ilustram o caso. Seja determinar a árvore geradora de menor peso do grafo da Figura 5.1. Os passos do algoritmo estão indicados na Figura 5.2. A operação inicial consiste em definir uma floresta com 6

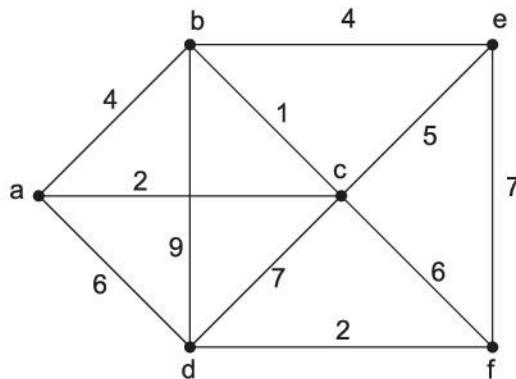


Figura 5.1: Um grafo com pesos nas arestas

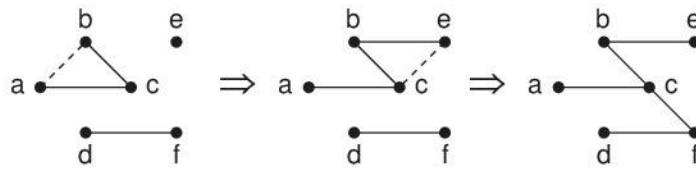
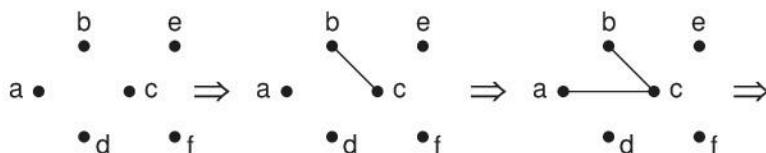


Figura 5.2: Algoritmo de árvore geradora mínima

árvores triviais, com rótulos  $\{a, b, c, d, e, f\}$ , respectivamente. Em seguida, as arestas são consideradas em ordem não decrescente de pesos. A aresta  $(b, c)$  de peso 1 é selecionada em primeiro lugar. Em seguida as arestas  $(d, f)$  e  $(a, c)$ , de peso 2, numa ordem arbitrária. A aresta  $(a, b)$  de peso 4 é rejeitada em sequência, pois produziria o ciclo  $a, b, c, a$ . A aresta  $(b, e)$  de peso 4 é então selecionada, sendo  $(c, e)$ , em seguida, rejeitada. Finalmente,  $(c, f)$  é escolhida, completando a fusão da floresta numa só árvore. Observe que a aresta  $(a, d)$  poderia ter sido escolhida ao invés de  $(c, f)$ , pois ambas possuem o mesmo peso. Nesse caso seria obtida uma outra solução, diferente, para o problema.

Os lemas seguintes atestam a correção do método. O Lema 5.1 assegura que a estrutura obtida pelo algoritmo é de fato uma árvore geradora de  $G$ , enquanto o 5.2 comprova que a mesma possui peso mínimo.

### Lema 5.1

Seja  $G(V, E)$  um grafo conexo. Seja  $T(V, E_T)$  o subgrafo obtido pela aplicação do algoritmo guloso acima. Então  $T$  é uma árvore geradora de  $G$ .

**Prova** É imediato que  $T$  é um subgrafo gerador acíclico. Resta mostrar que  $T$  é conexo. Suponha o contrário e sejam  $T_1, T_2$  duas árvores distintas da floresta  $T$ . Então a primeira aresta  $(v, w)$  da sequência ordenada de arestas tal que  $v$  está em  $T_1$  e  $w$  em  $T_2$ , quando adicionada a  $T$  não pode produzir ciclo. Consequentemente  $(v, w)$  não pode ser rejeitada pelo algoritmo guloso, o que leva à conclusão de que  $G$  é desconexo, uma contradição. ■

### Lema 5.2

Seja  $G(V, E)$  um grafo conexo e  $T(V, E_T)$  a árvore geradora obtida pela aplicação do algoritmo guloso acima. Então  $T$  possui peso mínimo.

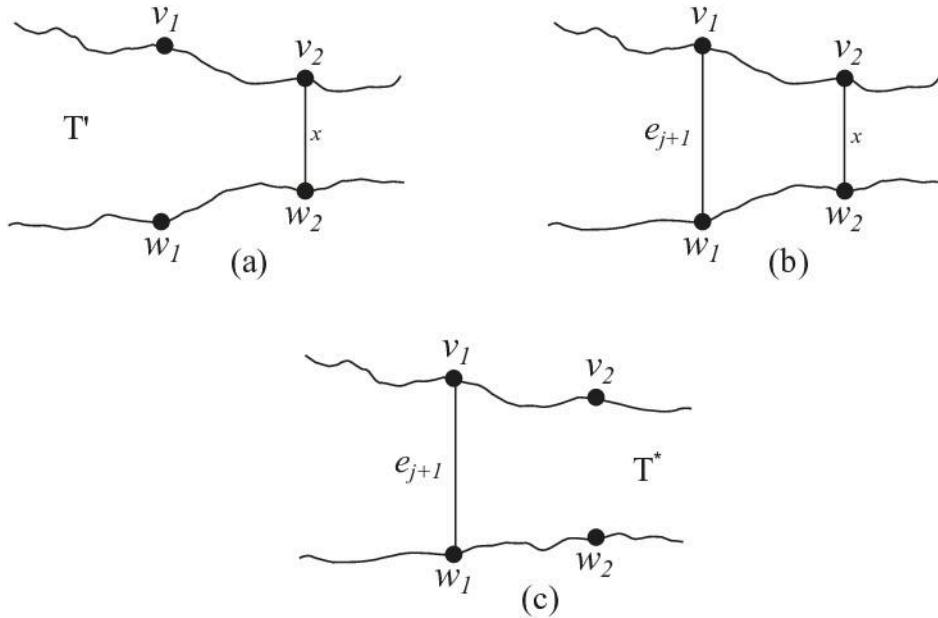


Figura 5.3: Prova do Lema 5.2

**Prova** Sejam  $e_1, \dots, e_{n-1}$  as arestas de  $T$ , na ordem em que foram consideradas pelo algoritmo, isto é,  $d(e_1) \leq \dots \leq d(e_{n-1})$ . Suponha, por contradição, que  $T$  não seja mínima. Dentre todas as árvores geradoras mínimas, seja  $T'(V, E_{T'})$  aquela que contém as arestas  $e_1, \dots, e_j$ , para o maior  $j$  possível. Obviamente  $j < n - 1$ . Adicione a aresta  $e_{j+1} \in E_T$  à árvore  $T'$ . O ciclo que então se forma necessariamente contém uma aresta  $x \neq e_i$ ,  $1 \leq i \leq j + 1$ , caso contrário não seria um ciclo. Sabe-se que  $d(e_{j+1}) \leq d(x)$ , caso contrário a aresta  $x$  teria sido selecionada pelo algoritmo guloso, no lugar de  $e_{j+1}$ . Se  $d(e_{j+1}) < d(x)$  então a árvore geradora  $T^*(V, [E_{T'} - \{x\}] \cup \{e_{j+1}\})$  possui peso menor do que  $T'$ , uma contradição (Figura 5.3). Se  $d(e_{j+1}) = d(x)$ , então a árvore  $T^*$  também é mínima e contém as arestas  $e_1, e_2, \dots, e_j, e_{j+1}$  o que contradiz a hipótese de que  $j$  era o menor possível. Logo,  $T$  deve ser mínima. ■

Observe que se o grafo  $G$  não for conexo, a aplicação do algoritmo guloso produziria uma floresta geradora de peso mínimo. Note também que através de um procedimento análogo poderia ser obtida a árvore geradora de peso máximo. Essas observações conduzem ao Algoritmo 5.1.

---

**Algoritmo 5.1:** Árvore geradora mínima

---

**Dados:**  $G(V, E)$  conexo,  $V = \{1, \dots, n\}$

definir conjuntos  $S_j := \{v_j\}$ ,  $1 \leq j \leq n$  e  $E_T = \emptyset$

seja  $e_1, \dots, e_m$  as arestas de  $G$ , ordenadas segundo pesos não decrescentes.

**para**  $j := 1, \dots, m$  **efetuar**

  seja  $v, w$  o par de vértices extremos de  $e_j$

**se**  $v, w$  pertencem respectivamente a conjuntos  $S_p, S_q$  disjuntos **então**

$S_p := S_p \cup S_q$

    eliminar  $S_q$

$E_T := E_T \cup \{e_j\}$

---

Observe que na descrição acima as árvores que compõem a floresta (a ser fundida na árvore geradora solução) são identificadas pelos conjuntos de vértices que as compõem, respectivamente. A adição de uma aresta à solução corresponde a uma união dos conjuntos que contêm os vértices extremidades dessa aresta. Ao final do processo, o conjunto  $E_T$  contém a solução do problema.

$n$	0	1	2	3	4	5	6	7	8	9
$F(n)$	0	1	1	2	3	5	8	13	21	34

Figura 5.4: Sequência de Fibonacci, para  $n \leq 9$ 

Para ordenar as arestas de  $G$ , é necessário  $O(m \log n)$  tempo. A determinação dos conjuntos  $S_p, S_q$  aos quais pertencem respectivamente os vértices  $v, w$ , bem como as operações de união (disjunta) podem ser efetuadas em  $O(\log n)$  para cada aresta considerada. O algoritmo portanto tem complexidade  $O(m \log n)$ .

## 5.4 Programação Dinâmica

Seja  $P$  um problema dado. Às vezes, é possível decompor  $P$  em um certo número de subproblemas  $P'$  de natureza igual ou semelhante a  $P$ , de tal modo que, resolvendo cada um desses subproblemas, obtém-se uma solução para  $P$ . Naturalmente, cada um dos subproblemas  $P'$  deve ser de tamanho menor do que  $P$ . Frequentemente, esses subproblemas não são necessariamente diferentes. Isto é, a solução de  $P$  pode depender de  $m_1$  problemas  $P'_1, m_2$  problemas  $P'_2$ , e assim por diante. Para obter uma solução para  $P$ , nesse caso, seria obviamente mais interessante resolver uma única vez, ao invés de  $m_i$  vezes cada um desses subproblemas  $P'_i$ . Para tanto, uma ideia seria resolver o subproblema  $P'_i$ , na primeira vez que esse fosse considerado, armazenando-se o resultado em uma tabela. Assim sendo, em todas as ocasiões subsequentes em que fosse necessário resolver novamente  $P'_i$ , bastaria uma consulta à tabela em questão, para obter o resultado desejado. Essa técnica, essencialmente, constitui o que se denomina *programação dinâmica*.

A ideia da programação dinâmica pode ser sintetizada pela observação de que ela constitui um processo de recursão, no qual dois procedimentos recursivos idênticos são computados uma única vez.

Para que  $P$  possa ser resolvido através desta técnica é necessário que a decomposição de  $P$  nos subproblemas  $P'$  seja de natureza relativamente simples. Usualmente,  $P$  e  $P'$  estão relacionados através de fórmulas de recorrência. Um outro aspecto é a definição da tabela de armazenamento dos resultados dos subproblemas  $P'$ . Obviamente, essa tabela deve ser definida, de modo a tornar simples o acesso aos seus resultados.

Mencione-se também que, frequentemente, o problema  $P$  e os subproblemas  $P'$  não são de natureza exatamente igual. Pode acontecer que  $P'$  seja mais geral do que  $P$ . Mesmo assim, devido ao tamanho mais reduzido de  $P'$ , a técnica pode produzir algoritmos eficientes.

Um exemplo simples de programação dinâmica é o de determinar uma *sequência de Fibonacci*. Esta consiste em uma sequência de inteiros, cujos primeiros dois elementos são 0 e 1, respectivamente. Subsequentemente, cada inteiro da sequência é definido como a soma dos dois imediatamente anteriores. Denotando por  $F(0)$  o primeiro elemento da sequência, por  $F(1)$  o seu segundo elemento a assim por diante, o seguinte seria um processo natural para sua determinação.

```
se  $n \leq 1$  então  $F(n) := n$ 
caso contrário  $F(n) := F(n - 1) + F(n - 2)$ 
```

Assim, o problema de se determinar  $F(n)$  foi decomposto nos subproblemas de determinar  $F(n - 1)$  e  $F(n - 2)$ , respectivamente. Para calcular  $F(n - 1)$ , com  $n - 1 > 1$ , necessita-se dos resultados dos subproblemas  $F(n - 2)$  e  $F(n - 3)$ . A ideia então é calcular cada um destes, exatamente uma vez, armazenando o resultado em uma tabela. Esta seria consultada cada vez que o mesmo subproblema fosse reconsiderado. Nesse caso, a tabela pode ser constituída simplesmente de um vetor de tamanho  $n + 1$ , sendo  $F(n)$  o elemento da sequência que se deseja calcular. A Figura 5.4 ilustra o método para  $n = 10$ . Observa-se que cada elemento da sequência é calculado em tempo constante, iniciando-se a computação de  $n = 0$  e prosseguindo em ordem crescente. Logo, a complexidade do processo é  $O(n)$ .

Na próxima seção será apresentada uma aplicação mais elaborada dessa técnica.

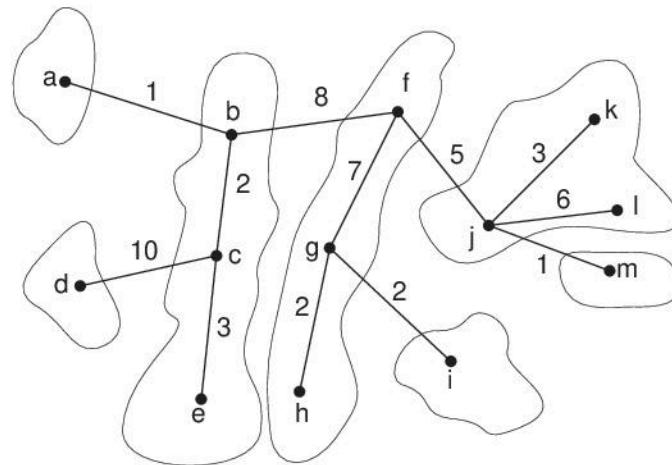


Figura 5.5: Um particionamento de uma árvore

## 5.5 Particionamento de Árvores

Nesta seção examina-se um algoritmo de programação dinâmica para o problema de *particionamento de árvores*.

Seja  $T(V, E)$  uma árvore em que a cada aresta  $e \in E$  existe um peso  $d(e)$  associado. Os pesos podem ser números reais não negativos. Seja dado também um inteiro positivo  $k$ . O problema consiste em particionar  $T$  em subárvores (disjuntas)  $S_1(V_1, E_1), S_2(V_2, E_2), \dots, S_t(V_t, E_t)$ , de tal modo que (i) cada subárvore possua no máximo  $k$  vértices e (ii) o somatório dos pesos das arestas que conectam subárvores diferentes seja mínimo. Isto é, para  $i = 1, \dots, t$

- (i)  $|V_i| \leq k$ , e
- (ii)  $\sum_{e \notin \bigcup E_i} d(e)$  mínimo

Observe que o particionamento de  $T$  em subárvores é perfeitamente determinado por um particionamento  $\{V_1, \dots, V_t\}$  de seus vértices,  $\bigcup V_i = V$  e  $V_i \cap V_j = \emptyset$ , para  $i \neq j$ . Além disso, vale também ser relembrado que cada  $S_i$  é uma subárvore, portanto conexa. O conjunto das subárvores que particionam  $T$  será chamado, simplesmente, *partição*. Cada subárvore da partição é uma *parte* (*de tamanho*  $\leq k$ ). Uma aresta que conecta subárvores diferentes da partição, isto é, uma aresta  $e \in E - \bigcup E_i$ ,  $1 \leq i \leq t$ , é denominada *aresta de corte*. A soma dos pesos das arestas de corte de uma partição é o *custo da partição*. Nesses termos, o problema consiste em determinar uma partição de  $T$  de custo mínimo, em que cada parte possui até  $k$  vértices. Esta partição será denominada *ótima*.

Seja  $T$  uma árvore e  $P = \{S_1(V_1, E_1), S_2(V_2, E_2), \dots, S_t(V_t, E_t)\}$  uma partição de  $T$ . O *peso da parte*  $S_j$  de  $P$  é o somatório dos pesos das arestas que formam  $S_j$ . Ou seja,  $\sum_{e \in E_j} d(e)$ . O *peso da partição*  $P$  é o somatório dos pesos das partes de  $P$ , isto é,  $\sum_{1 \leq j \leq t} \sum_{e \in E_j} d(e)$ . Observe que em relação à partição  $P$ , uma dada aresta ou pertence a alguma parte de  $P$  ou, caso contrário, é uma aresta de corte. Portanto, a partição  $P$  induz também um particionamento das arestas de  $T$  nas seguintes duas classes disjuntas: (a) arestas que pertencem a alguma subárvore de  $P$ , e (b) arestas de corte.

Naturalmente, o somatório dos pesos de todas as arestas de (a) e (b), ou seja, das arestas de  $T$ , é um dado constante. Logo, minimizar o custo da partição (pesos de (b)) é equivalente a maximizar o peso da partição (pesos de (a)). Este fato será utilizado pelo algoritmo a ser descrito. O presente problema também pode ser enunciado como: determinar uma partição de  $T$  de peso máximo, em que cada parte possui até  $k$  vértices.

A Figura 5.5 ilustra uma árvore  $T$ , com pesos indicados nas arestas, na qual foi realizado um particionamento em subárvores, com  $k = 3$ . Este corresponde ao seguinte particionamento de vértices:

$$\{(a), (d), (b, c, e), (f, g, h), (k, j, l), (i), (m)\}$$

As arestas de corte da partição indicada são  $(a, b)$ ,  $(d, c)$ ,  $(b, f)$ ,  $(f, j)$ ,  $(g, i)$  e  $(j, m)$ . A soma dos pesos de todas as arestas é 50. O peso dessa partição é  $(0) + (0) + (2 + 3) + (7 + 2) + (3 + 6) + (0) + (0) = 23$ . Seu custo

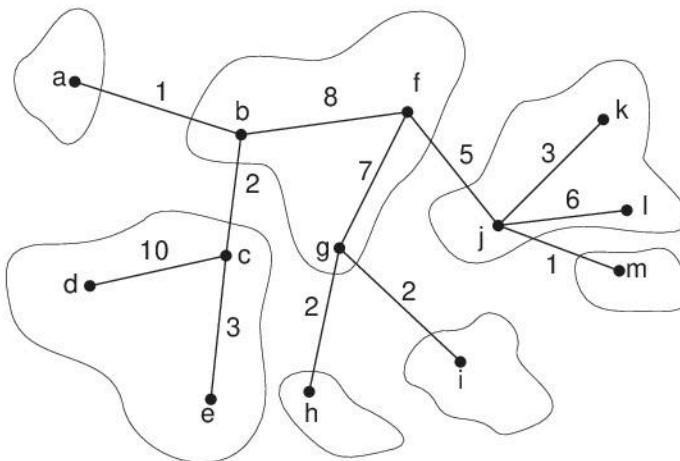


Figura 5.6: Um particionamento ótimo

$é 1 + 10 + 8 + 5 + 2 + 1 = 27$ . A partição do exemplo não é ótima. A Figura 5.6 ilustra uma outra partição da mesma árvore  $T$ , com custo 13. Pode ser verificado que esta última é de fato ótima.

Seja agora dada a árvore  $T(V, E)$ , onde cada aresta  $e = (v, w)$  possui um peso  $d(v, w)$ . Descreve-se a seguir um algoritmo para determinar uma partição de  $T$ , de peso máximo (ou seja, custo mínimo), onde cada parte possui até  $k$  vértices, sendo  $k$  um número inteiro dado.

O algoritmo utiliza técnicas de programação dinâmica. A árvore  $T$  será considerada como enraizada, sendo sua raiz um vértice arbitrário de  $T$ . O problema será dividido em  $n = |V|$  subproblemas, um para cada vértice.

Para cada  $v \in V$  o subproblema de  $v$  consiste em determinar, para cada  $j$ ,  $1 \leq j \leq k$ , a partição ótima  $P(v, j)$  da subárvore de raiz  $v$  e tal que a parte dessa partição que contém  $v$  possui exatamente  $j$  vértices. Observe que cada subproblema é de natureza mais geral do que o problema original. Ou seja, para cada vértice  $v$  considera-se a subárvore  $T_v$ , de raiz  $v$ , para a qual são definidos  $k$  subproblemas. Em cada um desses, o objetivo é encontrar a partição de peso máximo, na qual a subárvore que contém a raiz  $v$  possui exatamente  $j$  vértices,  $1 \leq j \leq k$ .

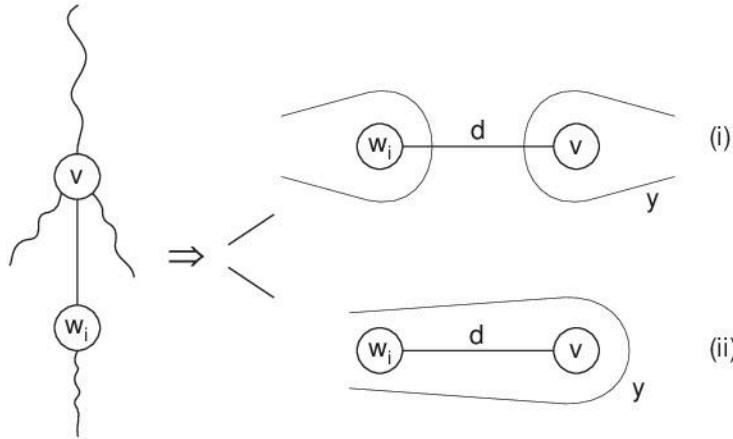
Seja  $p(v, j)$  o peso da partição  $P(v, j)$ . Nestes termos, os subproblemas de  $v$  consistem em determinar os valores de  $p(v, j)$  para  $1 \leq j \leq k$ . Observe que se  $T_v$  possui menos do que  $j$  vértices então  $P(v, j)$  não é definida. Nesse caso, por convenção,  $p(v, j) = -\infty$ .

O progresso da computação é das folhas para a raiz. Isto é, inicialmente são considerados os subproblemas relativos às folhas. Estes são triviais. Se  $u$  é uma folha de  $T$ , então  $P(u, 1) = \{(u)\}$  e  $P(u, j)$  não é definida para  $j > 1$ . Consequentemente,  $p(u, 1) = 0$  e  $p(u, j) = -\infty$ ,  $j > 1$ . No caso geral, um subproblema relativo a um vértice  $v$  somente pode ser considerado após terem sido resolvidos todos os subproblemas relativos aos filhos de  $v$ . O processo se desenvolve até que a raiz seja atingida, isto é, até que todos os subproblemas sejam resolvidos para todos os vértices. Observe que a solução do problema de particionamento de  $T$  decorre diretamente das soluções dos subproblemas da raiz  $r$  de  $T$ . Isto é, o peso da partição ótima de  $T$  é igual ao valor máximo dos pesos  $p(r, j)$ , para  $1 \leq j \leq k$ .

Considere agora resolver os subproblemas para a subárvore  $T_v$  de raiz  $v$ . Sejam  $w_1, \dots, w_f$  os filhos de  $v$  em  $T$ , numa ordem arbitrária. Supõem-se já resolvidos os subproblemas, para cada filho  $w_i$  de  $v$ . Seja calcular o valor  $p(v, j)$ . De um modo geral, a partição ótima  $P(v, j)$  pode apresentar duas situações diferentes, em relação a cada filho  $w_i$  de  $v$ :

- (i)  $v$  e  $w_i$  pertencem a partes diferentes, ou
- (ii)  $v$  e  $w_i$  pertencem à mesma parte.

No primeiro caso, a aresta  $(v, w_i)$  é necessariamente aresta de corte, logo, não deve ser computada no peso  $p(v, j)$ . No segundo,  $(v, w_i)$  está incluída em alguma parte de  $P(v, j)$ , logo  $d(v, w)$  deve constituir parcela de  $p(v, j)$ . Veja Figura 5.7.

Figura 5.7: Construção de  $P(v, j)$ 

Se  $v$  é um vértice qualquer de  $T$ , então é útil denotar a partição ótima da subárvore  $T_v$  por  $p(v, 0)$ . Ou seja,  $p(v, 0) = \max_{1 \leq j \leq k} \{p(v, j)\}$ . O lema seguinte fornece uma relação entre  $p(v, j)$  e  $p(w_i, h)$ , para todo filho  $w_i$  de  $v$ , e algum  $h$ ,  $1 \leq h \leq k$ .

### Lema 5.3

Seja  $P(v, j)$  a partição ótima da subárvore  $T_v$ , cuja parte  $y$  que contém  $v$  possui  $j$  vértices,  $1 \leq j \leq k$ . Seja  $w$  um filho de  $v$ . Então os vértices de  $T_w$  induzem em  $P(v, j)$  uma partição  $Q$  de peso igual a  $p(w, h)$ , sendo  $h$  o número de vértices de  $T_w$  pertencentes a  $y$ .

**Prova** Seja  $q$  o peso de  $Q$ . Se  $q < p(w, h)$  então a substituição em  $P(v, j)$  de  $Q$  por  $P(w, h)$  produz uma partição de peso maior do que  $p(v, j)$ , uma contradição. Por outro lado,  $q > p(w, h)$  já é uma contradição. Logo,  $q = p(w, h)$ . ■

Para formar a partição  $P(v, j)$ , consideram-se o vértice  $v$ , bem como as partições  $P(w_i, h)$ ,  $1 \leq i \leq f$ ,  $1 \leq h \leq k$  e formam-se composições compatíveis com as situações (i) ou (ii) anteriores. Para cada partição  $P(w_i, h)$  utiliza-se exatamente uma dentre as alternativas seguintes.

- (i) A partição  $P(w_i, h)$  é simplesmente adicionada a  $P(v, j)$ .
- (ii) A parte que contém  $w_i$  de  $P(w_i, h)$  é incorporada à parte  $y$  que contém  $v$  de  $P(v, j)$ . As demais partes de  $P(w_i, h)$  são adicionadas a  $P(v, j)$ .

Para avaliar o peso de  $P(v, j)$ , observe que se  $w_i$  não foi incluído em  $y$ , então a aresta  $(v, w_i)$  é aresta de corte. Logo seu peso não deve ser computado em  $p(v, j)$ . Quando  $w_i$  pertence a  $y$ , a aresta  $(v, w_i)$  também se encontra em  $y$ . Isto significa que  $d(v, w_i)$  é uma parcela de  $p(v, j)$ . Para um inteiro  $h$  defina:

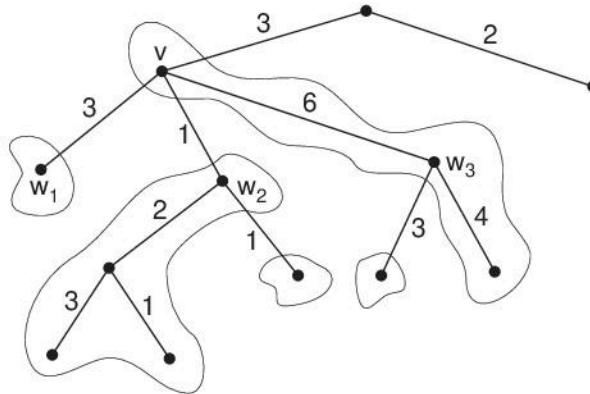
$$\begin{aligned}\alpha(h) &= 1 \text{ se } h \neq 0 \\ \alpha(h) &= 0 \text{ se } h = 0.\end{aligned}$$

Logo,  $p(v, j)$  será igual ao total máximo de

$$\sum_{i=1}^f p(w_i, h_i) + \alpha(h_i)d(v, w_i),$$

para todas as composições que satisfazem  $h_1 + h_2 + \dots + h_f = j - 1$ . Observe que um filho  $w_i$  de  $v$  pertence a  $y$  se e somente se  $h_i \neq 0$ .

No exemplo da Figura 5.8, seja  $k = 4$  e considere calcular o peso  $p(v, 3)$  da partição mais pesada de  $T_v$ , cuja parte que contém  $v$  possui exatamente 3 vértices. Supõem-se resolvidos os subproblemas para  $w_1, w_2$  e  $w_3$ , filhos

Figura 5.8: Construção de  $P(v, 3)$ , sendo  $k = 4$ 

	$j$				
	0	1	2	3	4
$p(w_1, j)$	0	0	$-\infty$	$-\infty$	$-\infty$
$p(w_2, j)$	6	4	5	5	6
$p(w_3, j)$	7	0	4	7	$-\infty$

Figura 5.9: Dados para o cálculo de  $p(v, 3)$ 

de  $v$ . Isto é, supõem-se já calculados os valores  $p(w_i, j)$ , para  $1 \leq i \leq 3$  e  $0 \leq j \leq 4$ , os quais são dados na Figura 5.9.

A Figura 5.10 fornece as composições possíveis que satisfazem  $h_1 + h_2 + h_3 = 2$ , juntamente com os pesos das partições correspondentes, obtidas através da expressão

$$\sum_{i=1}^3 p(w_i, h_i) + \alpha(h_i)d(v, w_i).$$

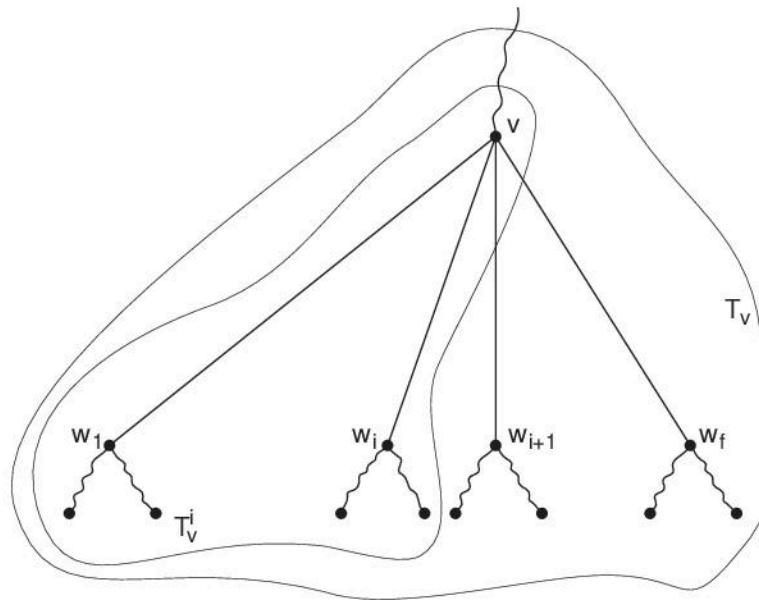
Note que  $h_i$  representa o número de elementos de  $T_{w_i}$  pertencentes à parte que contém  $v$ , da partição candidato a  $P(v, 3)$ . Naturalmente,  $P(v, 3)$  será a de maior peso dentre elas. Examinando-se o cálculo conclui-se que  $p(v, 3) = 16$ , obtido pela composição  $(h_1, h_2, h_3) = (0, 0, 2)$ . Esta partição está indicada na Figura 5.8.

A aplicação do processo anterior, evidentemente, resolve o problema. Calcula-se  $p(v, j)$ , para todo  $j$ ,  $0 \leq j \leq k$  e todo vértice  $v$  da árvore  $T$ . Com estes valores, o peso do particionamento ótimo da árvore será então  $p(r, 0)$ , onde  $r$  é a raiz de  $T$ . Contudo,  $p(v, j) = \max_{\sum h_i=j-1} \left\{ \sum_{i=1}^f p(w_i, h_i) + \alpha(h_i)d(v, w_i) \right\}$  implicaria em calcular o somatório tantas vezes quantas são as composições do inteiro  $j - 1$ , para cada vértice  $v$ . Este fato tornaria impraticável a utilização do método. Torna-se então necessário empregar uma estratégia adicional para produzir um processo mais eficiente.

Seja  $v$  um vértice da árvore enraizada  $T$ , com filhos  $w_1, \dots, w_f$ ,  $f \geq 1$ . Defina  $T_v^i = T_v - \bigcup_{l < l \leq f} T_{w_l}$ . Isto é,  $T_v^i$  é a subárvore parcial de raiz  $v$ , contendo as subárvores  $T_{w_1}, \dots, T_{w_i}$  (Figura 5.11). A ideia consiste em utilizar

$h_1$	$h_2$	$h_3$	$\sum_{i=1}^3 p(w_i, h_i) + \alpha(h_i)d(v, w_i)$
0	0	2	$(0 + 0) + (6 + 0) + (4 + 6) = 16$
0	2	0	$(0 + 0) + (5 + 1) + (7 + 0) = 13$
2	0	0	$(-\infty + 3) + (6 + 0) + (7 + 0) = -\infty$
0	1	1	$(0 + 0) + (4 + 1) + (0 + 6) = 11$
1	0	1	$(0 + 3) + (6 + 0) + (0 + 6) = 15$
1	1	0	$(0 + 3) + (4 + 1) + (7 + 0) = 15$

Figura 5.10: Cálculo de  $p(v, 3)$

Figura 5.11: A subárvore parcial  $T_v^i$ 

os particionamentos ótimos de  $T_v^{i-1}$  e  $T_{w_i}$ , para o cálculo correspondente a  $T_v^i$ ,  $i > 1$ . O lema seguinte fornece uma relação entre essas partições.

#### Lema 5.4

Seja  $v$  um vértice de  $T$ , com filhos  $w_1, \dots, w_f$ ,  $f \geq 1$ . Para  $i > 1$ , seja  $P^i(v, j)$  a partição ótima de  $T_v^i$ , de peso  $p^i(v, j)$  e cuja parte  $y$  que contém  $v$  possui  $j$  vértices. Então

- (i) os vértices de  $T_v^{i-1}$  induzem em  $P^i(v, j)$  uma partição de peso exatamente igual a  $p^{i-1}(v, g)$ , sendo  $g > 0$  o número de vértices de  $T_v^{i-1}$  em  $y$ .
- (ii) os vértices de  $T_{w_i}$  induzem em  $P^i(v, j)$  uma partição de peso igual a  $p(w_i, h)$ , sendo  $h \geq 0$  o número de vértices de  $T_{w_i}$  em  $y$ .

**Prova** Caso contrário, a troca da partição induzida respectivamente por (i)  $P^{i-1}(v, g)$  ou (ii)  $P(w_i, h)$  aumentaria o peso de  $P^i(v, j)$ , uma contradição. ■

Portanto, a parte  $y$  de  $P^i(v, j)$  possui  $g > 0$  vértices de  $P^{i-1}(v, g)$  e  $h \geq 0$  de  $P(w_i, h)$ , sendo  $g + h = j$ . Supondo já calculados os valores  $p^{i-1}(v, g)$  e  $p(w_i, h)$ , obtém-se  $p^i(v, j)$  através de

$$p^i(v, j) = \max_{g+h=j, g>0} \{ p^{i-1}(v, g) + p(w_i, h) + \alpha(h)d(v, w_i) \}$$

Para  $i = 1$ , a expressão acima também pode ser utilizada, adotando-se a convenção  $p^0(v, 1) = 0$  e  $p^0(v, j) = -\infty$ , para  $j > 1$ .

Sendo  $f$  o número de filhos de  $v$ ,  $p^f(v, j) = p(v, j)$  e portanto o cálculo iterativo de  $p^i(v, j)$  para valores crescentes de  $i$  conduz à solução do problema. Consequentemente, não é mais necessário determinar as composições de  $j - 1$ , para computar  $p(v, j)$ . Na inicialização do processo, defina

$$\begin{aligned} p(v, j) &= 0, \text{ se } j \leq 1 \\ p(v, j) &= -\infty, \text{ caso contrário.} \end{aligned}$$

Essas condições iniciais já resolvem os subproblemas das folhas. Denote por  $w_1, \dots, w_f$  os filhos de  $v$ , numa ordem qualquer, e suponha resolvidos os subproblemas para cada  $w_i$ . Isto é, são conhecidos  $p(w_i, h)$ ,  $0 \leq h \leq k$ . Para resolver o subproblema de  $v$ , a ideia consiste em supor que a subárvore  $T_v$  é construída passo a passo, através da incorporação a  $T_v$  das subárvore  $T_{w_i}$  para  $1 \leq i \leq f$ . No passo inicial,  $T_v$  consiste unicamente em  $v$ . Após

cada incorporação de  $T_{w_i}$  a  $T_v$ , corrigem-se os valores de  $p(v, j)$  obtidos. No caso geral, todas as subárvore  $T_{w_1}, T_{w_2}, \dots, T_{w_{i-1}}$ , já foram incorporadas a  $T_v$ , isto é, o particionamento  $P^{i-1}(v, j)$  já foi calculado,  $0 \leq j \leq k$ . A incorporação de  $T_{w_i}$  a  $T_v$  é realizada mediante a aplicação do Lema 5.4, obtendo-se assim os valores de  $p^i(v, j)$  correspondentes. A formulação seguinte descreve o algoritmo.

---

**Algoritmo 5.2:** Particionamento de árvores

---

**Dados:** árvore  $T(V, E)$ , com peso não negativo  $d(v, w)$  em cada aresta  $(v, w)$  e um inteiro  $k > 0$

considerar  $T$  como árvore enraizada, de raiz arbitrariamente escolhida

**para**  $v \in V$  **efetuar**

**para**  $j = 0, \dots, k$  **efetuar**

**se**  $j \leq 1$  **então**

$p(v, j) := 0$

**caso contrário**

$p(v, j) := -\infty$

    marcar as folhas de  $T$  e desmarcar seus vértices interiores

**enquanto** houver vértices desmarcados **efetuar**

$v :=$  algum vértice desmarcado, cujos filhos  $w_1, \dots, w_f$  são todos marcados

**para**  $i = 1, \dots, f$  **efetuar**

**para**  $j = 1, \dots, k$  **efetuar**

$q(v, j) := \max_{g+h=j, g>0} \{p(v, g) + p(w_i, h) + \alpha(h)d(v, w_i)\}$

**para**  $j = 1, \dots, k$  **efetuar**

$p(v, j) := q(v, j)$

$p(v, 0) := \max_{l \leq j \leq k} \{p(v, j)\}$

    marcar  $v$

---

Os valores de  $p(v, j)$  são armazenados em uma tabela, de  $n = |V|$  linhas e  $k + 1$  colunas, a qual é preenchida conforme indicado pelo Algoritmo 5.2. O processo termina quando o valor  $p(r, 0)$  da tabela é calculado, onde  $r$  é a raiz da árvore. Este valor é a solução do problema. Além da tabela mencionada, necessita-se de um vetor adicional  $q(v, j)$  com  $k$  elementos, para armazenar temporariamente os valores de  $p(v, j)$ , para cada  $j$ ,  $1 \leq j \leq k$ . Esse vetor é posteriormente reutilizado, para cada vértice considerado. Logo, a complexidade de espaço é  $O(nk)$ . Para determinar a complexidade de tempo, observe que a computação de  $q(v, j)$  é, sem dúvida, a mais crítica do processo. O cálculo de cada  $q(v, j)$  corresponde à determinação do máximo do conjunto

$$\begin{aligned} \{ & p(v, 1) + p(w_i, j-1) + d(v, w_i), \\ & p(v, 2) + p(w_i, j-2) + d(v, w_i), \\ & \dots, \\ & p(v, j-1) + p(w_i, 1) + d(v, w_i), \\ & p(v, j) + p(w_i, 0) \}, \end{aligned}$$

o qual possui  $j$  elementos. Cada elemento pode ser computado em tempo constante, pois envolve valores já calculados e armazenados na tabela. Logo, cada  $q(v, j)$  pode ser calculado em tempo  $O(j)$ . Para cada vértice  $v$ ,  $q(v, j)$  é computado para  $j = 1, \dots, k$ . Logo, todos os  $p(v, i)$ , para um certo  $v$ , podem ser calculados em tempo  $O(k^2)$ . A complexidade de tempo é pois  $O(nk^2)$ . Observe que para  $k$  constante as complexidades de espaço e tempo, ambas, tornam-se lineares no número de vértices da árvore.

Como exemplo, seja particionar a árvore da Figura 5.12(a), com  $k = 3$ . Foi escolhido (arbitrariamente) como raiz o vértice  $g$ , como indica a Figura 5.12(b). Os vértices foram considerados na ordem  $a, b, c, d, e, f, g$  que satisfaz a condição requerida de que um vértice é considerado apenas quando todos os seus descendentes já o tiverem sido. O particionamento ótimo tem portanto peso igual a  $p(g, 0)$ , ou seja, 13. O cálculo encontra-se indicado na Figura 5.13.

Considere agora uma extensão do presente problema. A árvore dada é tal que, além dos pesos nas arestas, a cada vértice também é associado um certo peso, inteiro não negativo (os pesos nas arestas são números reais não

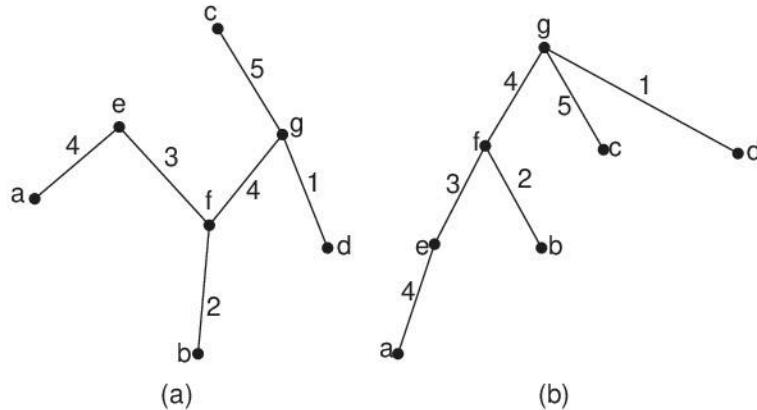


Figura 5.12: O exemplo correspondente à Figura 5.13

$v \ j$	0	1	2	3
$a$	0	0	$-\infty$	$-\infty$
$b$	0	0	$-\infty$	$-\infty$
$c$	0	0	$-\infty$	$-\infty$
$d$	0	0	$-\infty$	$-\infty$
$e$	4	0	4	$-\infty$
$f$	7	4	6	7
$g$	13	7	12	13

Figura 5.13: Tabela de solução do problema da Figura 5.12, com  $k = 3$ 

negativos). Sendo dado um inteiro  $k$ , deseja-se agora determinar uma partição  $P$  dos vértices da árvore, tal que a soma dos pesos dos vértices em cada parte de  $P$  seja menor ou igual a  $k$  e a soma dos pesos das arestas de corte seja mínima. Obviamente, o problema até agora discutido é um caso particular da mencionada extensão, em que os pesos dos vértices são todos unitários. Pode ser imediatamente verificado que o Algoritmo 5.2 (que obtém a partição ótima para o caso especial de pesos unitários) é também aplicável, com uma pequena alteração, para a extensão considerada. Observe que, agora, se  $Z(v)$  é o peso do vértice  $v$ , então em relação a qualquer partição, o peso da parte que contém  $v$  é maior ou igual a  $Z(v)$ . Portanto, neste caso, o algoritmo deve computar os valores de  $p(v, j)$ , para todo vértice  $v$  e para  $j = Z(v), Z(v) + 1, \dots, k$  (ao invés de  $j = 1, \dots, k$ , como no caso anterior). Note que  $k \geq Z(v)$ , caso contrário o problema não teria solução. A expressão da complexidade do algoritmo permanece a mesma, ou seja,  $O(nk^2)$ . Contudo, conforme será comentado no Capítulo 6, a complexidade perde a propriedade, nesta extensão, de poder ser expressa através de um polinômio cuja variável seja o tamanho dos dados de entrada.

Uma outra aplicação de programação dinâmica é apresentada na Seção 9.14.

## 5.6 Alteração Estrutural

Seja  $G(V, E)$  um grafo. A técnica de alteração estrutural consiste em aplicar algum tipo de operação a  $G$ , de modo a transformá-lo em outro grafo  $G'(V', E')$ . Suponha que se deseja resolver um problema  $P$ , no grafo  $G$ . Seja  $P'$  um problema relacionado a  $P$  e tal que se possa resolver  $P'$  em  $G'$ . Obviamente, se for possível extrapolar a solução de  $P$  em  $G$ , a partir da obtida de  $P'$  em  $G'$ , resolve-se o problema inicial desejado.

Naturalmente, o tipo de transformação a ser aplicada ao grafo dado pode ser de natureza qualquer e depende, intrinsecamente, do problema que se deseja resolver. Contudo, as seguintes transformações são comuns:

- (i) *Eliminação de vértices ou arestas:* Nessa transformação escolhe-se, de forma conveniente, um vértice  $v$ , ou aresta  $e$ , do grafo  $G(V, E)$ , o qual é retirado de  $G$ . O grafo alterado é simplesmente  $G'(V - \{v\}, E')$ ,

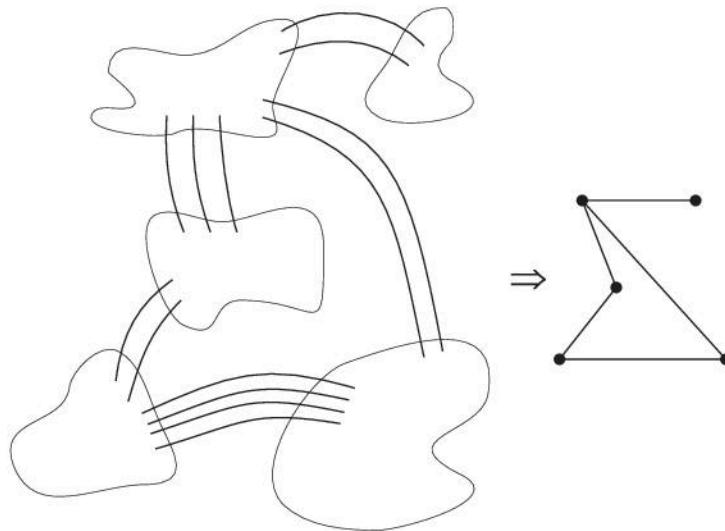


Figura 5.14: Operação de condensação

ou respectivamente  $G'(V, E - \{e\})$ . Como a transformação é simples, frequentemente os problemas que admitem solução através dessa técnica também são de natureza simples. Observe que o tamanho de  $G'$  é menor, do que o de  $G$ , fato que pode contribuir para tornar o problema  $P'$  mais simples do que  $P$ .

- (ii) *Adição de vértices ou arestas:* Essa transformação é inversa, em relação à anterior. Ao invés de se retirar, acrescentam-se novos vértices ou arestas ao grafo dado.
- (iii) *Condensação:* A técnica de condensação consiste em transformar certos componentes distintos do grafo  $G$  dado em um único. Esses componentes, em geral, são subconjuntos de vértices ou arestas. A transformação corresponde, pois, a uma operação de *identificação*. A aplicação dessa operação produz um novo grafo  $G'$ , de tamanho menor do que  $G$ . A Figura 5.14 ilustra a técnica.

Observe que, de um modo geral, a operação de condensação pode ser reduzida a um conjunto de operações mais elementares, de eliminação e adição de vértices ou arestas.

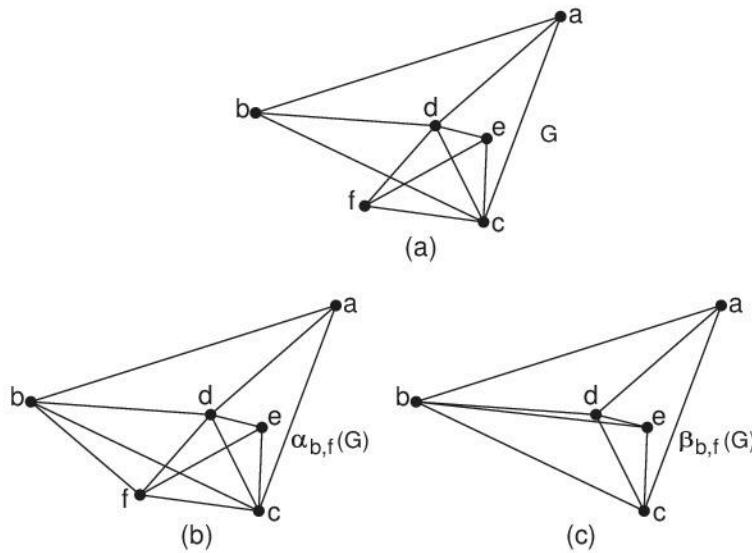
A seção seguinte apresenta uma aplicação da técnica de alteração estrutural, onde o grafo dado é transformado em dois outros, o primeiro mediante uma adição de aresta e o segundo através do emprego da condensação.

## 5.7 Número Cromático

Nesta seção, descreve-se um algoritmo para determinar o número cromático de um grafo. O algoritmo constitui um exemplo de aplicação da técnica de alteração estrutural.

Seja  $G(V, E)$  o grafo não direcionado dado. Se  $G$  for o grafo completo  $K_n$ , o cálculo de seu número cromático é trivial, sendo igual a  $n$ . Caso contrário, existem dois vértices distintos  $v$  e  $w$ , não adjacentes. A ideia é efetuar duas alterações estruturais diferentes em  $G$ , utilizando os vértices  $v, w$ . Denota-se por  $\alpha_{v,w}(G)$  o grafo obtido a partir de  $G$ , pela adição da aresta  $(v, w)$ . Ou seja,  $\alpha_{v,w}(G)$  é o grafo  $G = (V, E \cup \{(v, w)\})$ , enquanto que  $\beta_{v,w}(G)$  é construído a partir de  $G$ , pela condensação dos vértices  $v, w$  em um único vértice  $z$ , eliminando-se as arestas paralelas, que porventura possam ter se formado. Observe que a formação de arestas paralelas acontece quando existir algum vértice de  $G$  simultaneamente adjacente a  $v$  e  $w$ . Por exemplo, seja  $G$  o grafo da Figura 5.15(a). Então os grafos desenhados nas Figuras 5.15(b) e 5.15(c) correspondem a  $\alpha_{b,f}(G)$  e  $\beta_{b,f}(G)$ , respectivamente.

A ideia do algoritmo para a obtenção do número cromático  $\chi(G)$  de um grafo  $G(V, E)$  é simples. Se  $G$  for completo, evidentemente  $\chi(G) = |V|$ . Caso contrário, existe um par de vértices não adjacentes  $v, w$ . Determinam-se os grafos  $\alpha_{v,w}(G)$  e  $\beta_{v,w}(G)$ . O número cromático de  $G$  é então mínimo entre os números cromáticos dos grafos  $\alpha_{v,w}(G)$  e  $\beta_{v,w}(G)$ . Para a obtenção dos números cromáticos de  $\alpha_{v,w}(G)$  e  $\beta_{v,w}(G)$ , respectivamente, aplica-se

Figura 5.15: Os grafos  $\alpha$  e  $\beta$ 

esta mesma estratégia, de forma recursiva. Observe que, através dessa recursão, cada grafo será eventualmente transformado num grafo completo, para o qual o número cromático é facilmente computado.

A correção do método baseia-se no teorema seguinte.

### Teorema 5.1

Seja  $G$  um grafo não completo e  $v, w$  um par de vértices não adjacentes de  $G$ . Então o número cromático  $\chi(G)$  satisfaz:

$$\chi(G) = \min\{\chi(\alpha_{v,w}(G)), \chi(\beta_{v,w}(G))\}$$

**Prova** Suponha uma coloração ótima  $C$  de  $G$ . Se  $v, w$  possuem cores diferentes em  $C$ , então a aresta  $(v, w)$  pode ser adicionada a  $G$ , sem alterar  $C$ . Nesse caso,  $\chi(G) = \chi(\alpha_{v,w}(G))$ . Se  $v, w$  possuem cores iguais  $c_i$  em  $C$ , então  $v, w$  podem ser condensados em um único vértice  $z$ . Atribui-se a  $z$  a cor  $c_i$ , mantendo-se as cores de  $C$ , para as demais. Nesse caso,  $\chi(G) = \chi(\beta_{v,w}(G))$ . Logo,  $\chi(G)$  deve ser o menor entre  $\chi(\alpha_{v,w}(G))$  e  $\chi(\beta_{v,w}(G))$ . ■

Observe ainda que pela aplicação recursiva do Teorema 5.1 pode-se afirmar que o número cromático de um grafo  $G$  é igual ao número de vértices do menor grafo completo que pode ser obtido através de operações sucessivas de formação dos grafos  $\alpha$  e  $\beta$ , a partir de  $G$ .

---

### Algoritmo 5.3: Determinação do número cromático $\chi$ de um grafo

---

**Dados:** grafo  $G(V, E)$

**Procedimento** COR( $G$ )

seja  $n_G$  o número de vértices de  $G$

se  $G$  é completo **então**

$$\chi := \min\{\chi, n_G\}$$

**caso contrário**

encontrar um par de vértices  $v, w$  não adjacentes em  $G$

$$\text{COR}(\alpha_{v,w}(G))$$

$$\text{COR}(\beta_{v,w}(G))$$

$$\chi := n$$

$$\text{COR}(G)$$


---

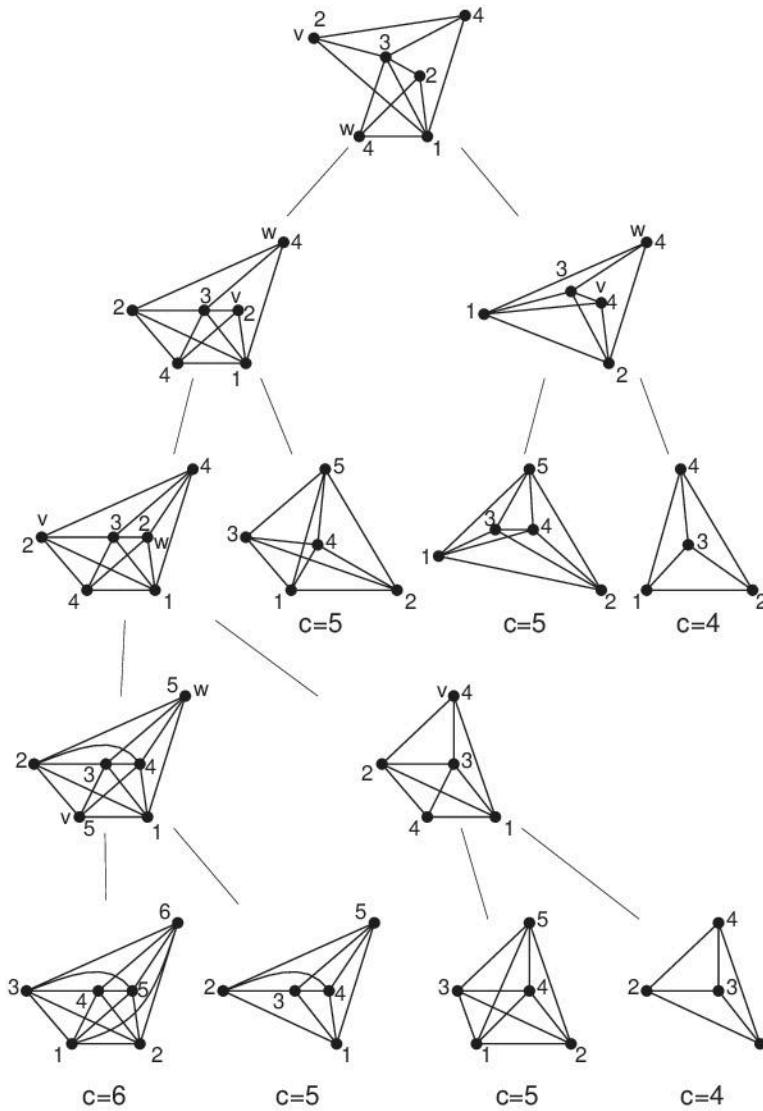


Figura 5.16: Determinação do número cromático

Observe que o Algoritmo 5.3 constrói implicitamente uma árvore estritamente binária  $T$ , onde cada vértice da árvore corresponde a um grafo. O grafo  $G$  dado está associado à raiz de  $T$ . Se  $G'$  é um grafo não completo associado a um vértice genérico de  $T$ , então seus filhos esquerdo e direito correspondem, respectivamente, aos grafos  $\alpha_{v,w}(G')$  e  $\beta_{v,w}(G')$ , sendo  $v, w$  um par de vértices não adjacentes. As folhas da árvore são necessariamente grafos completos. Naturalmente, o número cromático de  $G$  é precisamente igual ao mínimo número de vértices, dentre os grafos completos associados às folhas. O algoritmo constrói a árvore em pré-ordem, onde a subárvore esquerda de cada vértice é totalmente construída, antes de iniciar a da direita. A Figura 5.16 mostra a árvore obtida para determinar o número cromático do grafo ilustrado na Figura 5.15(a). Observe que o menor grafo completo, correspondente às folhas, possui 4 vértices. Logo o número cromático procurado é igual a 4.

Se for desejado obter a coloração ótima, além do número cromático, pode-se empregar o Algoritmo 5.3, com uma ligeira variação. Efetua-se a coloração de cada grafo da árvore  $T$ , de baixo para cima. A prova do Teorema 5.1 fornece uma indicação de como obter a mencionada coloração. Como uma folha de  $T$  é um grafo completo  $K_p$ , sua coloração é trivial (usa-se uma cor diferente para cada um dos  $p$  vértices). Suponha que os grafos  $\alpha_{v,w}(G')$  e  $\beta_{v,w}(G')$  já tenham sido coloridos, para algum grafo  $G'$ , correspondente a um vértice de  $T$ . Se  $\chi(\alpha_{v,w}(G')) < \chi(\beta_{v,w}(G'))$ , então a coloração ótima de  $G'$  será exatamente igual a de  $\alpha_{v,w}(G')$ . Se  $\chi(\alpha_{v,w}(G')) > \chi(\beta_{v,w}(G'))$

então atribuir a ambos os vértices  $v, w$  de  $G'$  a mesma cor do vértice  $z$  de  $\beta_{v,w}(G)$ , onde  $z$  é a identificação de  $v, w$ . Se  $\chi(\alpha_{v,w}(G'))$  e  $\chi(\beta_{v,w}(G'))$  forem iguais, aplica-se qualquer das duas operações.

O número de vértices da árvore  $T$  pode ser exponencial em relação ao número de vértices do grafo  $G$ . Consequentemente, este algoritmo possui complexidade exponencial.

## 5.8 Programas em Python

Esta seção contém implementações dos algoritmos formulados neste capítulo. As implementações seguem as descrições gerais apresentadas nos Capítulos 1 e 2.

### 5.8.1 Algoritmo 5.1: Árvore Geradora Mínima

Na implementação do Algoritmo 5.1, utilizamos a estrutura de dados da união disjunta, definida na classe `UniaoDisjunta`. Na Linha 4, definimos `S` como uma união disjunta vazia. As Linhas 5–6 inserem cada vértice em `S` inicialmente. As Linhas 8–12 correspondem à implementação da definição da sequência de arestas  $e_1, \dots, e_m$  no algoritmo. Note que, na implementação, a ordenação é feita através da função `sort` própria de listas de Python, que recebe como parâmetro `key` uma função que mapeia cada objeto da lista (aresta) a uma chave, que será usada internamente como chave de comparação para a ordenação. No caso, definimos uma função sem nome (através do comando `lambda`) que atribui como chave de cada aresta `e` o peso `e.w` desta aresta. O restante da implementação é tradução direta do algoritmo, observando-se que  $S_p \leftarrow S_p \cup S_q$  e a eliminação de  $S_q$  corresponde à chamada `S.Une(v, w)` e que `S.Conjunto(v)` retorna um inteiro que corresponde a um identificador do conjunto que contém `v`.

Programa 5.1: Árvore geradora mínima

```

1 #Algoritmo 5.1: árvore geradora mínima
2 #Dado: Grafo conexo G em lista de adjacência, E(G) rotulado com inteiro w
3 def ArvoreGerMinima(G):
4     S = UniaoDisjunta(G.n);
5     for v in G.V():
6         S.Insere(v)
7     ET = []; E = []
8     for v in G.V():
9         for w_no in G.N(v, IterarSobreNo=True):
10            if w_no.Viz < v:
11                E.append(w_no.e)
12    E.sort(key=lambda e: e.w)
13    for e in E:
14        v,w = e.v1,e.v2
15        if S.Conjunto(v) != S.Conjunto(w):
16            S.Une(v,w)
17            ET.append(e)
18    return ET
19
20 class UniaoDisjunta:
21     def __init__(self, n):
22         self.Pai, self.n = [None]*(n+1), 0
23
24     def Une(self, u, v):
25         Ru = self.Conjunto(u)
26         Rv = self.Conjunto(v)
27         if Ru != Rv:
28             if -self.Pai[Ru] < -self.Pai[Rv]:
29                 self.Pai[Ru], self.Pai[Rv] = Rv, self.Pai[Rv] + self.Pai[Ru]
30             else:
31                 self.Pai[Rv], self.Pai[Ru] = Ru, self.Pai[Rv] + self.Pai[Ru]
```

```

32
33     def Conjunto(self, u):
34         if self.Pai[u] > 0:
35             self.Pai[u] = self.Conjunto(self.Pai[u])
36             return self.Pai[u]
37         else:
38             return u
39
40     def Insere(self, u):
41         self.Pai[self.n+1], self.n = -1, self.n+1

```

### 5.8.2 Algoritmo 5.2: Particionamento de Árvore

O Programa 5.2 corresponde à implementação do Algoritmo 5.2. A implementação segue diretamente o algoritmo, observando-se que a sequência de visitas aos vértices da árvore enraizada, que deve ser feito a partir das folhas até a raiz, é implementada como uma busca em profundidade simplificada (não é necessário considerar arestas de retorno).

Programa 5.2: Particionamento de árvores

```

#Algoritmo 5.2: Particionamento de árvores
#Dados: árvore T ( $V, E$ ), com peso não negativo  $d(v, w)$  em cada aresta  $(v, w)$  e um inteiro  $k > 0$ 
def ParticionamentoArvore(T,d,k):
    #considerar T como árvore enraizada, de raiz r arbitrariamente escolhida
    r = 1
    p = [None]*(T.n+1)
    for v in range(1,T.n+1):
        p[v] = [None]*(k+1)
    for v in range(1,T.n+1):
        for j in range(k+1):
            if j <= 1:
                p[v][j] = 0
            else:
                p[v][j] = float("-inf")
    def Visitar(T,d,v,paiv):
        for i in T.N(v):
            if i != paiv:
                Visitar(T,d,i,v)
        for i in T.N(v):
            if i != paiv:
                q_v = [None]*(k+1)
                for j in range(1,k+1):
                    def alfa(h):
                        return 0 if h==0 else 1
                    q_v[j] = max([p[v][g]+p[i][j-g]+alfa(j-g)*d[v][i] for g in range(1,j+1)])
                for j in range(1,k+1):
                    p[v][j] = q_v[j]
    p[v][0] = max([p[v][j] for j in range(1,k+1)])
    Visitar(T,d,r,None)

    return p[r][0]

```

### 5.8.3 Algoritmo 5.3: Número Cromático

O Programa 5.3 corresponde diretamente à implementação do Algoritmo 5.3.

Programa 5.3: Determinação do número cromático  $\chi$  de um grafo

```

1 #Algoritmo 5.3: Determinação do número cromático X de um grafo
2 #Dados: grafo G
3 def COR(G):
4     global X
5     nG = G.n
6     if G.m == nG*(nG-1)/2:
7         X = min(X,nG)
8     else:
9         (v,w) = NaoAdj(G)
10        COR(alfa(v,w,G))
11        COR(beta(v,w,G))
12
13 def alfa(v,w,G):
14     H = GrafoMatrizAdj(orientado=False)
15     H.DefinirN(G.n)
16     for (x,y) in G.E():
17         H.AdicionarAresta(x,y)
18     H.AdicionarAresta(v,w)
19     return H
20
21 def beta(v,w,G):
22     def NovoId(t,w):
23         return t if t<w else t-1
24     H = GrafoMatrizAdj(orientado=False)
25     H.DefinirN(G.n-1) #w será removido, identificado com v
26     for (x,y) in G.E():
27         if x != w and y != w:
28             xlin,ylin=NovoId(x,w),NovoId(y,w)
29             H.AdicionarAresta(xlin,ylin)
30     for x in G.N(w):
31         xlin,ylin=NovoId(x,w),NovoId(v,w)
32         if not H.SaoAdj(xlin,ylin):
33             H.AdicionarAresta(xlin,ylin)
34     return H
35
36 def NaoAdj(G):
37     for v in G.V():
38         for w in G.V():
39             if v != w and not G.SaoAdj(v,w):
40                 return (v,w)

```

## 5.9 Exercícios

- 5.1 Considere a seguinte variação do algoritmo guloso para determinação da árvore geradora mínima  $T(V, E_T)$  de um grafo  $G(V, E)$ . “O primeiro passo é incluir em  $E_T$  a aresta de maior peso de  $E$ . No passo geral, incluir em  $E_T$  a aresta de maior peso de  $E$ , que possua exatamente um extremo incidente a alguma aresta já incluída em  $E'_T$ ”. Formular uma implementação desse algoritmo. Obter também a sua complexidade.

- 5.2 Caracterizar os grafos, com pesos nas arestas, para os quais todas as árvores geradoras mínimas são isomórfas entre si.
- 5.3 Caracterizar os grafos, com pesos nas arestas, para os quais todas as árvores geradoras mínimas são diferentes entre si.
- 5.4 Deseja-se calcular o elemento  $F(n)$  da sequência de Fibonacci. Para tanto, considera-se um processo semelhante ao do descrito na Seção 5.4, exceto que o resultado de cada subproblema é recalculado toda vez que utilizado (no algoritmo do texto, cada resultado é computado uma única vez e armazenado numa tabela para uso posterior). Qual a complexidade do novo processo?
- 5.5 Estender o Algoritmo 5.2 para encontrar também a partição ótima da árvore, além de seu peso.
- 5.6 Formular uma implementação do algoritmo para a extensão do problema de particionamento de árvores, mencionada no final da Seção 5.5. Nessa, os dados são um inteiro  $k$  e uma árvore  $T(V, E)$ , na qual cada aresta possui um peso real e cada vértice um peso inteiro, não negativos. O objetivo consiste em particionar  $T$  em subárvores disjuntas, de tal modo que a soma dos pesos dos vértices de cada subárvore é  $\leq k$ , e a soma dos pesos das arestas de todas as subárvores é máxima.
- 5.7 Considere o problema de particionamento de uma árvore  $T$ , conforme definido na Seção 5.5. A seguinte é uma tentativa de obter a partição ótima  $P$  de  $T$ , através de um algoritmo guloso. “O passo inicial é definir  $P = \{e\}$ , onde  $e$  é a aresta de maior peso de  $T$ . O passo geral consiste em incluir em  $P$  a aresta  $e'$  de maior peso ainda não considerada, de modo que a subárvore de  $P$  a qual  $e'$  pertence possua no máximo  $k$  vértices, após essa inclusão.” Mostrar, através de um exemplo, que esse algoritmo não está correto.
- 5.8 Seja o problema de *particionamento de grafos*. Os dados são um inteiro  $k$  e um grafo  $G(V, E)$ , com pesos não negativos nas arestas. O objetivo consiste em particionar  $V$  em subconjuntos  $V_1, \dots, V_q$ , de tal modo que cada  $|V_i| \leq k$  e o somatório dos pesos das arestas com extremos em subconjuntos  $V_i$  diferentes seja mínima. A seguinte é uma tentativa de resolver este problema. “Obter uma árvore geradora máxima  $T$ , de  $G$ , utilizando uma variação do Algoritmo 5.1. Em seguida, encontrar o particionamento ótimo de  $T$ , mediante a aplicação do Algoritmo 5.2. Considerar esse particionamento como o procurado para  $G$ .” Mostrar, através de um exemplo, que este algoritmo não está correto.
- 5.9 Qual seria a complexidade do Algoritmo 5.2, se o valor de cada  $p(v, j)$  fosse recalculado toda vez que utilizado (o mencionado algoritmo o calcula uma única vez e armazena seu resultado numa tabela, para uso posterior)?
- 5.10 Dê exemplo de um grafo  $G$ , para o qual

$$\chi(\alpha_{v,w}(G)) < \chi(\beta_{v,w}(G)),$$

sendo  $v, w$  um par arbitrário de vértices não adjacentes em  $G$ .

- 5.11 Caracterizar os grafos  $G$  para os quais a árvore binária  $T$  produzida pela aplicação do Algoritmo 5.3 é tal que todo grafo  $G'$ , correspondente a um vértice interior de  $T$ , satisfaz  $\chi(\alpha_{v,w}(G')) \leq \chi(\beta_{v,w}(G'))$ . Qual a complexidade do processo de obtenção do número cromático para esta classe de grafos?
- 5.12 A aplicação do Algoritmo 5.3 a um grafo  $G(V, E)$  produz uma árvore binária  $T$ , cujas folhas correspondem a grafos completos  $K_p$  tais que existe pelo menos uma folha em  $T$  para cada  $p$ ,  $\chi(G) \leq p \leq |V|$ . Provar ou dar contra-exemplo.
- 5.13 Obter a expressão da complexidade do Algoritmo 5.3 para determinação do número cromático de um grafo.

#### 5.14 UVA Online Judge 11228

Escreva um algoritmo para o seguinte problema:

Graflândia é um país pobre com muitas cidades, mas sem estradas. O governo pretende interligar todas as cidades por rodovias ou ferrovias. Cidades de um mesmo estado serão conectadas por rodovias e a interligação entre estados será por ferrovias. Quando duas cidades têm distância inferior a um valor  $d$ , elas estão no mesmo estado. Dados  $d, n$  (número de cidades) e as posições geográficas (cartesianas planares) das cidades, determinar os comprimentos totais de rodovias e de ferrovias a serem construídas, de tal forma que se tenha custo total mínimo (o custo é proporcional ao tamanho das estradas).

#### 5.15 UVA Online Judge 11857

Escreva um algoritmo para o seguinte problema:

Muitos fabricantes de automóveis estão fabricando carros elétricos, que exigem baterias pesadas e caras. Desta forma, precisam projetar com cuidado as baterias e, portanto, o alcance desses carros. Sua tarefa é calcular o alcance mínimo a ser conseguido com as baterias, tal que seja possível viajar entre quaisquer duas cidades do continente. Sabe-se que em todas as cidades existem estações de recargas das baterias. São dadas as  $n$  cidades, as  $m$  estradas entre elas, todas bidirecionais, e os comprimentos das mesmas.

#### 5.16 UVA Online Judge 10397

Escreva um algoritmo para o seguinte problema:

Há muitos prédios em construção no campus da Universidade de Waterloo. Você foi contratado como programador e deve assegurar que cada prédio seja conectado aos demais, direta ou indiretamente, através da rede de cabos. Cabos devem conectar diretamente os edifícios. Os prédios antigos já estão conectados entre si. Dados  $n$  prédios, sua posição geográfica e a rede de cabos já existente, você deve calcular a quantidade mínima necessária de cabos para interligar os novos prédios aos já existentes.

## 5.10 Notas Bibliográficas

O Algoritmo 5.1, para determinar a árvore geradora mínima de um grafo  $G(V, E)$ , é de Kruskal (1956). A variação descrita no Exercício 5.1 corresponde ao algoritmo de Prim (1957). Uma outra formulação do uso do algoritmo guloso para resolver esse problema foi realizada por Dijkstra (1959). Posteriormente, algoritmos de complexidade  $O(m \log \log n)$  foram apresentados em Yao (1975) e Cheriton e Tarjan (1976). Algoritmos para resolver o problema da árvore geradora ótima de um grafo foram desenvolvidos ainda anteriormente aos descritos neste capítulo. Com efeito, já em 1926, Otakar Boruvka

elaborou um algoritmo para o problema, com o intuito de aplicá-lo na construção de uma rede elétrica na cidade de Moravia (Boruvka 1926). O algoritmo se inicia por escolher tentativamente para cada vértice do grafo, a aresta de menor peso a ele incidente. Entre 1926 e 1965, diversos outros autores descreveram algoritmos para o problema de árvore geradora ótima que são essencialmente similares ao de Boruvka. Um artigo reportando a história deste último algoritmo foi publicado por Nesetril, Milkova e Nesetrilova (2001). O algoritmo de menor complexidade existente até agora é de Chazelle (2000), que é quase linear, requerendo  $O(m\alpha(m, n))$  passos, onde  $\alpha(m, n)$  é o inverso da função de Ackerman, a qual em termos práticos se comporta como constante. Este algoritmo, de certa forma, usa também a técnica de Boruvka. O algoritmo guloso pode ser estudado através de matroides (Lawler (1976)). A técnica de programação dinâmica é empregada há algum tempo. Com efeito, um livro específico sobre o assunto foi publicado ainda em 1957 (Bellman (1957)). O Algoritmo 5.2, de particionamento de árvores, incorporando a extensão do Exercício 5.6, é de Lukes (1974). Um algoritmo de complexidade polinomial para o caso em que os pesos das arestas são unitários e os dos vértices arbitrários (situação inversa à do Algoritmo 5.1) foi apresentado por Hadlock (1974). O problema de particionamento de grafos, Exercício 5.8, foi também tratado por Lukes (1975). Aproximações para o problema são discutidas em Schrader (1981). O Algoritmo 5.3 é de Zykov (1949). Uma implementação deste algoritmo foi também realizada por Corneil e Graham (1973).

Página deixada intencionalmente em branco

---

# FLUXO MÁXIMO EM REDES

---

## 6.1 Introdução

O Capítulo 6 é dedicado ao estudo do fluxo máximo em redes. Este tópico é fundamental para a solução de diversos problemas de áreas diferentes, notadamente a otimização combinatória. Pois uma grande variedade de problemas podem ser resolvidos mediante sua transformação em um caso de fluxo máximo em redes. Além disso, os algoritmos de fluxo máximo constituem exemplos didáticos em complexidade computacional.

Na próxima seção são apresentadas as definições e propriedades básicas. O Teorema do Fluxo Máximo-Corte Mínimo, fundamental na teoria, é o assunto seguinte. Nas Seções de 6.4 a 6.7 são descritos algoritmos para resolver o problema do fluxo máximo. O primeiro deles possui complexidade exponencial, enquanto os demais são polinomiais com complexidades respectivamente decrescentes, segundo a sequência de apresentação.

## 6.2 O Problema do Fluxo Máximo

Uma *rede* é um digrafo  $D(V, E)$  em que a cada aresta  $e \in E$  está associado um número real positivo  $c(e)$  denominado *capacidade* da aresta  $e$ . Suponha que  $D$  possua dois vértices especiais e distintos  $s, t \in V$  chamados *origem* e *destino*, respectivamente, com as seguintes propriedades: o primeiro é uma fonte que alcança todos os vértices. Enquanto o destino é um sumidouro alcançado também por todos. Um *fluxo*  $f$  de  $s$  a  $t$  em  $D$  é uma função que a cada aresta  $e \in E$  associa um número real não negativo  $f(e)$  satisfazendo às seguintes condições:

- (i)  $0 \leq f(e) \leq c(e)$ , para toda aresta  $e \in E$
- (ii)  $\sum_{w_1} f(w_1, v) = \sum_{w_2} f(v, w_2)$ , para todo vértice  $v \neq s, t$ .

A primeira condição apresentada simplesmente indica que o fluxo em cada aresta não ultrapassa o valor de sua capacidade. A segunda significa que o fluxo se *conserva* em cada vértice  $v \neq s, t$ . Isto é, o somatório dos fluxos das arestas convergentes a  $v$  é igual ao das divergentes de  $v$ . Este somatório é denominado *valor* do fluxo em  $v$ . Por analogia, o somatório dos fluxos das arestas divergentes de  $s$  e o das convergentes a  $t$  são o *valor* do fluxo em  $s$  e o em  $t$ , respectivamente. O valor do fluxo na origem é denominado *valor* do fluxo na rede  $D$  e denotado por  $f(D)$ .

A Figura 6.1(a) ilustra um fluxo em uma rede. A capacidade e o fluxo em cada aresta estão indicados pelo par de números correspondentes, com o segundo entre parênteses. Por exemplo, a aresta  $(v_2, v_3)$  possui capacidade 2 e fluxo 1. O valor do fluxo no vértice  $v_2$  é 3, no destino  $t$  é 4 e na rede também é 4. A situação da rede da Figura 6.1(b) não corresponde a um fluxo. Pois o vértice  $v_4$  não satisfaz à condição (ii) anterior (há um total igual a 3 de fluxo convergente a  $v_4$  e um total 4 divergente). De um modo geral, se os valores  $f(e)$  não obedecerem às condições da definição anterior a atribuição  $f$  será chamada *fluxo ilegal*.

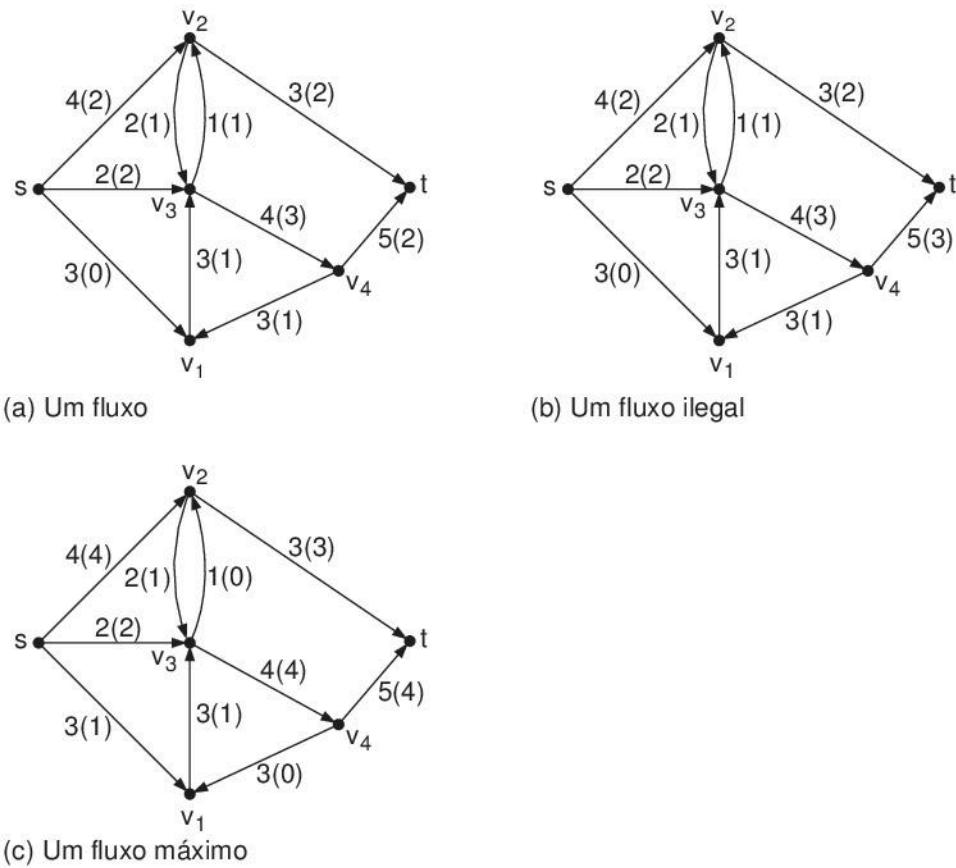


Figura 6.1: Fluxo em redes

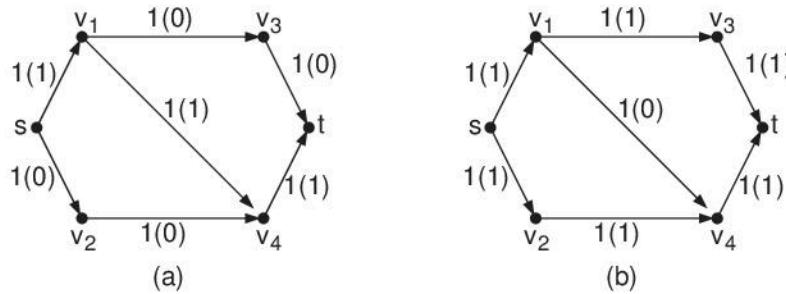


Figura 6.2: Fluxos maximal e máximo

O fluxo em uma rede possui certa analogia com o escoamento de água de uma origem  $s$  a um destino  $t$ , através de uma rede de tubulação. A capacidade de cada aresta corresponde à vazão máxima de água através da tubulação correspondente. O escoamento de água obedece às condições (i) e (ii) anteriores. Contudo, satisfaz também a restrições adicionais.

Naturalmente, o valor do fluxo em uma rede pode variar de um mínimo igual a zero até um certo máximo. Por exemplo, o valor do fluxo na Figura 6.1(c) é igual a 7, o qual é máximo para a sua rede. O *problema do fluxo máximo* consiste em, dada uma rede, determinar tal fluxo. Este será denominado *fluxo máximo*.

Seja  $f$  um fluxo em uma rede  $D(V, E)$ . Uma aresta  $e \in E$  é saturada quando  $f(e) = c(e)$ . Um vértice  $v \in V$  é saturado quando todas as arestas convergentes a  $v$  ou todas divergentes de  $v$  estão saturadas. No fluxo da Figura 6.1(c), a aresta  $(v_2, t)$  e o vértice  $v_4$  estão saturados. Um fluxo é maximal quando todo caminho de  $s$  a  $t$  em  $D$  contém alguma aresta saturada. Isto é, o valor de um fluxo maximal não pode ser aumentado simplesmente por acréscimos de fluxos em algumas arestas. Naturalmente, todo fluxo máximo é maximal. Contudo, a recíproca não é necessariamente verdadeira. Por exemplo, no fluxo da Figura 6.2(a), as arestas  $(s, v_1)$ ,  $(v_1, v_4)$  e  $(v_4, t)$  estão saturadas, bem como os vértices  $v_1$  e  $v_4$ . Este fluxo é maximal pois todo caminho de  $s$  a  $t$  contém uma dessas arestas (ou um desses vértices). Observe que ele não é máximo, pois possui valor 1 enquanto que o da Figura 6.2(b) possui valor 2.

Seja  $S \subseteq V$  um subconjunto de vértices tal que  $s \in S$  e  $t \notin S$ . Denote  $\bar{S} = V - S$ . Um *corte*  $(S, \bar{S})$  em  $D$  é o subconjunto das arestas de  $D$  que possuem uma extremidade em  $S$  e outra em  $\bar{S}$ . Assim sendo todo caminho da origem  $s$  ao destino  $t$  em  $D$  contém alguma aresta de  $(S, \bar{S})$ . Sejam

$$(S, \bar{S})^+ = \{(v, w) \in E | v \in S \text{ e } w \in \bar{S}\}$$

$$(S, \bar{S})^- = \{(v, w) \in E | v \in \bar{S} \text{ e } w \in S\}.$$

Define-se *capacidade*  $c(S, \bar{S})$  do corte  $(S, \bar{S})$  como o somatório das capacidades das arestas de  $(S, \bar{S})^+$ . Observe que o corte  $(S, \bar{S})$  inclui as arestas de  $(S, \bar{S})^-$ , mas essas não são utilizadas no cálculo de sua capacidade. Um *corte mínimo* é aquele que possui capacidade mínima.

Seja  $f$  um fluxo e  $(S, \bar{S})$  um corte em  $D$ . Então  $f(S, \bar{S})$  é o fluxo no corte  $(S, \bar{S})$  e definido como a diferença

$$f(S, \bar{S}) = \sum_{e \in (S, \bar{S})^+} f(e) - \sum_{e \in (S, \bar{S})^-} f(e)$$

Observe que, em geral,  $c(S, \bar{S}) \neq c(\bar{S}, S)$  e  $f(S, \bar{S}) \neq f(\bar{S}, S)$ .

Por exemplo, na rede da Figura 6.1(a), com  $S = \{s, v_2, v_3\}$  obtém-se  $\bar{S} = \{v_1, v_4, t\}$  e o corte  $(S, \bar{S}) = \{(s, v_1), (v_1, v_3), (v_3, v_4), (v_2, t)\}$ , no qual  $(S, \bar{S})^+ = \{(s, v_1), (v_3, v_4), (v_2, t)\}$ ,  $(S, \bar{S})^- = \{(v_1, v_3)\}$ ,  $c(S, \bar{S}) = c(s, v_1) + c(v_3, v_4) + c(v_2, t) = 10$  e  $f(S, \bar{S}) = f(s, v_1) + f(v_3, v_4) + f(v_2, t) - f(v_1, v_3) = f(S, \bar{S})^+ - f(S, \bar{S})^- = 4$ .

Observe que o valor do fluxo em uma rede é igual ao seu valor no corte  $(\{s\}, V - \{s\})$ . Na realidade este fato é ainda mais geral. O valor do fluxo em uma rede pode ser medido em qualquer corte, como indica o lema seguinte.

### Lema 6.1

Seja  $f$  um fluxo em uma rede  $D$  e  $(S, \bar{S})$  um corte em  $D$ . Então  $f(S, \bar{S}) = f(D)$ .

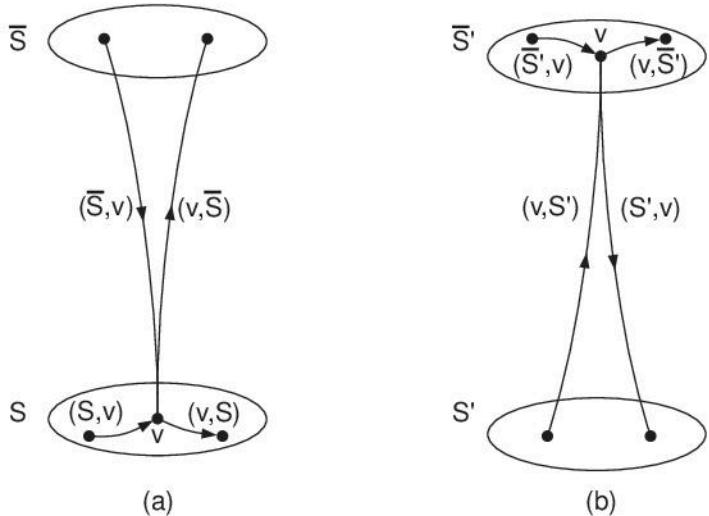


Figura 6.3: Prova do Lema 6.1

**Prova** Indução no tamanho de  $S$ . Se  $|S| = 1$ , então  $S = \{s\}$  e  $f(S, \bar{S}) = f(D)$  por definição. Para  $|S| > 1$  escolha algum  $v \in S$ ,  $v \neq s$ . Sejam  $(v, S)$  e  $(S, v)$  os conjuntos das arestas de  $E$  com uma extremidade em  $v$  em outra em  $S$  e dirigidas de  $v$  para  $S$  e de  $S$  para  $v$ , respectivamente. Denote  $S' = S - \{v\}$ . Sejam  $f(v, S)$  e  $f(S, v)$  os somatórios dos fluxos nas arestas de  $(v, S)$  e  $(S, v)$ , respectivamente. Observe que (Figura 6.3)  $f(S, \bar{S}) = f(S', \bar{S}') + f(S, v) - f(v, S) + f(\bar{S}, v) - f(v, \bar{S})$ . Pela condição (ii) da definição de fluxo,  $f(v, S) + f(v, \bar{S}) = f(S, v) + f(\bar{S}, v)$ . Logo,  $f(S, \bar{S}) = f(S', \bar{S}')$ . Pela hipótese de indução  $f(S', \bar{S}') = f(D)$ , o que completa a prova. ■

Note que, em particular,  $f(V - \{t\}, \{t\}) = f(\{s\}, V - \{s\}) = f(D)$ .

### 6.3 O Teorema do Fluxo Máximo – Corte Mínimo

Nesta seção inicia-se a abordagem ao problema do fluxo máximo. O objetivo inicial é estabelecer condições que permitam caracterizar um fluxo máximo. Como resultado principal, será formulado o Teorema do Fluxo Máximo – Corte Mínimo apresentado no final da seção.

Seja  $f$  um fluxo em uma rede  $D$ . O objetivo, no momento, é aumentar o valor de  $f$ , se possível. Cada aresta  $e$  pode receber um adicional de fluxo  $\leq c(e) - f(e)$ , o que talvez produza um aumento no valor de  $f$ . Uma aresta  $e$  tal que  $c(e) - f(e) > 0$  denomina-se *aresta direta*. É possível também que  $f$  não seja máximo e simultaneamente não seja possível aumentar  $f$  unicamente através de incrementos de fluxo em arestas diretas. Um fluxo maximal, mas não máximo, é um exemplo desta última afirmativa, pois neste caso não há caminho de  $s$  a  $t$  através unicamente de arestas diretas. Em consequência, há situações em que a única forma de aumentar  $f$  consiste em, além de incrementar o fluxo em algumas arestas, decrementá-lo em outras. Por exemplo, para aumentar o valor do fluxo na rede da Figura 6.2(a) torna-se necessário também decrementá-lo na aresta  $(v_1, v_4)$ . Naturalmente, uma aresta  $e$  pode receber um decreimento de fluxo positivo  $\leq f(e)$ . Se  $f(e) > 0$  então  $e$  é denominada *aresta contrária*. Na rede da Figura 6.1(a),  $(s, v_2)$  é aresta direta e contrária, enquanto  $(s, v_1)$  é direta e  $(s, v_3)$  é contrária.

Dados  $f$  e  $D(V, E)$  define-se a *rede residual*  $D'(f)$  como aquela em que o conjunto de vértices coincide com o de  $D$  e cujas arestas são obtidas pela seguinte construção:

“Se  $(v, w)$  é aresta direta de  $D$ , então  $(v, w)$  é aresta de  $D'$  também chamada *direta* e com capacidade  $c'(v, w) = c(v, w) - f(v, w)$ . Se  $(v, w)$  é aresta contrária de  $D$ , então  $(w, v)$  é aresta de  $D'$ , também chamada *contrária* e com capacidade  $c'(w, v) = f(v, w)$ .”

Em outras palavras, as capacidades das arestas de  $D'$  representam as possíveis variações de fluxo que as arestas de  $D$  podem sofrer, com o direcionamento de cada aresta indicando sua variação positiva ou negativa.

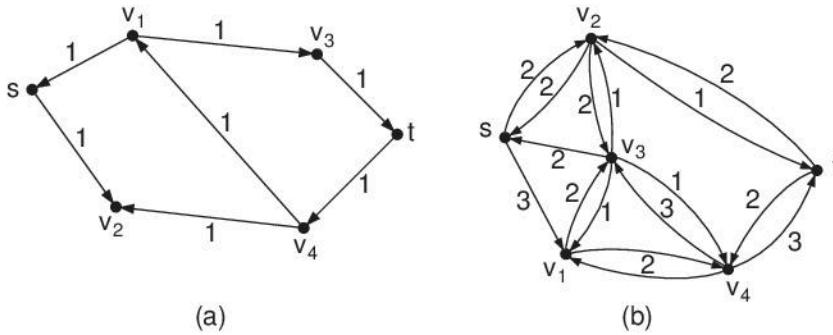


Figura 6.4: Redes residuais

Como exemplo, as redes das Figuras 6.4(a) e 6.4(b) são residuais respectivamente dos fluxos nas redes 6.2(a) e 6.1(a), respectivamente. As capacidades das arestas estão indicadas nas novas figuras.

Um caminho de  $s$  a  $t$  na rede residual  $D'(f)$  é denominado *aumentante* (ou *caminho de acréscimo de fluxo*) para  $f$ . Por exemplo, o caminho  $s, v_2, v_4, v_1, v_3, t$  na rede da Figura 6.4(a) é aumentante para o fluxo da Figura 6.2(a).

O lema seguinte descreve uma aplicação do conceito de rede residual ao problema do fluxo máximo.

### Lema 6.2

Seja  $f$  um fluxo em uma rede  $D(V, E)$  e  $D'$  a rede residual correspondente. Suponha que exista em  $D'$  um caminho aumentante  $v_1, \dots, v_k$  da origem  $s = v_1$  ao destino  $t = v_k$ . Então  $f(D)$  pode ser aumentado de um valor

$$F' = \min\{c'(v_j, v_{j+1}) \mid 1 \leq j < k\}.$$

**Prova** A construção seguinte obtém em  $D$  um novo fluxo de valor  $f(D) + F'$ . Para  $1 \leq j < k$ , se  $(v_j, v_{j+1})$  é aresta direta incrementar  $f(v_j, v_{j+1})$  de  $F'$ . Se  $(v_j, v_{j+1})$  é contrária, decrementar  $f(v_{j+1}, v_j)$  de  $F'$ . Em ambos os casos, o fluxo no vértice  $v_{j+1}$  recebeu  $F'$  unidades adicionais. Além disso, como  $F'$  é a menor dentre as capacidades das arestas do caminho aumentante, assegura-se que os novos fluxos nas arestas também satisfazem às condições da definição. Ou seja, a nova situação é também um fluxo  $f'$ . Pelo corte  $(\{s\}, V - \{s\})$  conclui-se que  $f'(D) = f(D) + [f'(s, v_2) - f(s, v_2)] = f(D) + F'$ . ■

Como exemplo, na rede residual (Figura 6.4(a)) do fluxo da Figura 6.2(a),  $s, v_2, v_4, v_1, v_3, t$  é um caminho aumentante no qual o valor mínimo entre as capacidades de suas arestas é 1. As arestas  $(s, v_2)$ ,  $(v_2, v_4)$ ,  $(v_1, v_3)$  e  $(v_3, t)$  são diretas, o que permite incrementar de 1 o fluxo em cada uma delas. A aresta  $(v_4, v_1)$  é contrária, o que permite decrementar de 1 em  $(v_1, v_4)$ . Com essas alterações, a Figura 6.2(a) se transforma na 6.2(b).

Como exemplo adicional, considere a rede residual da Figura 6.4(b), correspondente ao fluxo da 6.1(a). No caminho aumentante  $s, v_1, v_4, t$ , a capacidade mínima é 1. Assim sendo, pode-se incrementar de 1 o fluxo nas arestas  $(s, v_1)$  e  $(v_4, t)$  e diminuí-lo de 1 em  $(v_4, v_1)$ . Isto aumenta em uma unidade o valor do fluxo de 6.1(a). Se todo caminho entre a origem e o destino em uma rede  $D$  contiver uma certa aresta  $e$ , naturalmente o valor de qualquer fluxo em  $D$  não pode ultrapassar a capacidade  $c(e)$ . Isto é, esta aresta atua de forma semelhante a um “gargalo” para o fluxo. De uma forma mais geral, o valor de qualquer fluxo  $f$  em  $D$  não pode ultrapassar o valor de qualquer corte  $(S, \bar{S})$ . Pois  $f(D) = f(S, \bar{S}) = \sum_{e \in (S, \bar{S})^+} f(e) - \sum_{e \in (S, \bar{S})^-} f(e) \leq \sum_{e \in (S, \bar{S})^+} c(e) = c(S, \bar{S})$ .

Ou seja, o fluxo máximo em uma rede não pode ultrapassar a capacidade de seu corte mínimo. O teorema seguinte é fundamental em teoria de fluxo em redes. Ele afirma que esse limite superior é atingível.

### Teorema 6.1

O valor do fluxo máximo em uma rede  $D$  é igual à capacidade do corte mínimo de  $D$ .

**Prova** Seja  $f$  um fluxo máximo em  $D$ . Pelo observado anteriormente,  $f(D) \leq c_{\min}$ , onde  $c_{\min}$  é a capacidade do corte mínimo. Suponha  $f(D) < c_{\min}$  e seja  $D'$  a rede residual de  $f$ . Há duas possibilidades:

1º caso: Existe caminho aumentante para  $f$ . Esta situação contradiz o Lema 6.2.

2º caso: Não existe caminho aumentante. Seja  $S$  o conjunto de vértices alcançáveis em  $D'$  a partir de  $s$ . Naturalmente,  $s \in S$  e  $t \in \bar{S}$ . Caso contrário, se  $t \in S$  haveria caminho aumentante. Então

(i)  $D'$  não possui aresta direta de  $S$  para  $\bar{S}$ . Isto é,  $f(e) = c(e)$ , para cada  $e \in (S, \bar{S})^+$ .

(ii)  $D'$  não possui aresta contrária de  $S$  para  $\bar{S}$ . Isto é,  $f(e) = 0$ , para cada  $e \in (S, \bar{S})^-$ . Logo,  $f(D) = f(S, \bar{S}) = \sum_{e \in (S, \bar{S})^+} f(e) - \sum_{e \in (S, \bar{S})^-} f(e) = c(S, \bar{S})$ , o que contradiz  $f(D) < c_{\min}$ .

Logo,  $f(D) = c_{\min}$  ■

Os corolários seguintes são consequências diretas desse teorema.

### Corolário 6.1

Sejam  $(S, \bar{S})$  um corte e  $f$  um fluxo máximo uma rede  $D$ . Então  $(S, \bar{S})$  é mínimo se e somente se

- (i) toda aresta  $e \in (S, \bar{S})^+$  estiver saturada, e
- (ii) toda aresta  $e \in (S, \bar{S})^-$  satisfizer  $f(e) = 0$ .

### Corolário 6.2

Um fluxo  $f$  em uma rede  $D$  é máximo se e somente se não existir caminho aumentante para  $f$ .

No exemplo da Figura 6.2(a),  $(\{s\}, \{v_1, v_2, v_3, v_4, t\})$  é um corte mínimo de capacidade 2. Logo o fluxo máximo dessa rede possui valor 2 (Figura 6.2(b)). Na Figura 6.1(a). o corte  $(\{s, v_1, v_3, v_2\}, \{v_4, t\})$  é mínimo e possui capacidade 7. Isto permite concluir que o fluxo da Figura 6.1(c) é máximo.

## 6.4 Um Primeiro Algoritmo

Seja  $D(V, E)$  uma rede onde cada aresta  $e \in E$  possui capacidade  $c(e)$  inteira positiva. Nesta seção formula-se um primeiro algoritmo para determinar o fluxo máximo numa rede  $D$ .

A prova do Lema 6.2 é construtiva. Ela fornece, juntamente com o Teorema 6.1, o seguinte algoritmo:

---

#### Algoritmo 6.1: Fluxo máximo em uma rede (Ford e Fulkerson)

**Dados:** rede  $D(V, E)$ , com capacidades  $c(e)$  inteiras e positivas, para cada  $e \in E$ , origem  $s \in V$  e destino  $t \in V$

$F := 0$

**para**  $e \in E$  **efetuar**

$f(e) := 0$

    construir a rede residual  $D'(f)$

**enquanto** existir caminho  $v_1, \dots, v_k$  de  $s = v_1$  a  $t = v_k$  em  $D'$  **efetuar**

$F' := \min\{c'(v_j, v_{j+1}), |1 \leq j < k\}$

**para**  $j = 1, \dots, k-1$  **efetuar**

**se**  $(v_j, v_{j+1})$  é aresta direta **então**

$f(v_j, v_{j+1}) := f(v_j, v_{j+1}) + F'$

**caso contrário**

$f(v_{j+1}, v_j) := f(v_{j+1}, v_j) - F'$

$F := F + F'$

    construir a rede residual  $D'(f)$

---

Ou seja, no passo inicial define-se  $f(e) := 0$  para cada aresta  $e$  de  $D$ . O valor do fluxo  $F$  é portanto nulo. No passo geral, constrói-se a rede residual  $D'$  de  $D$ . Se houver caminho aumentante em  $D$  então  $F$  pode ser incrementado, conforme o Lema 6.2. Repete-se o processo. Se o caminho apresentado não existir, o fluxo é

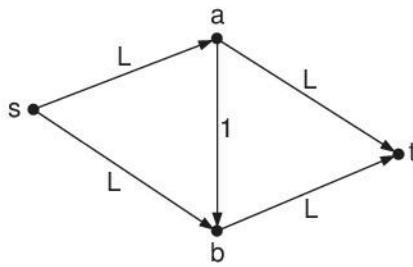


Figura 6.5: Um caso ruim para o Algoritmo 6.1

máximo. Ou seja, como as capacidades das arestas são números inteiros, os incrementos também o são. Portanto, o valor do fluxo se mantém inteiro no processo. Isto é, após um número finito de incrementos,  $F$  atinge o valor máximo igual à capacidade do corte mínimo de  $D$ . Para calcular a complexidade desse algoritmo é necessário estimar o número de iterações do bloco *enquanto*. Contudo esse número pode depender do valor  $F$  do fluxo máximo. Assim sendo, em um pior caso  $F$  pode ser exponencial no tamanho dos incrementos  $F'$ . Isto é, o número de iterações é  $\Omega(F)$ . Em cada iteração deve-se construir a rede residual  $D'$  e um caminho aumentante. Essas operações podem ser realizadas em tempo  $O(m)$ . O passo inicial também requer  $O(m)$  operações. Logo, a complexidade do algoritmo é  $O(m(F + 1))$ . Observe que  $F$  pode ser exponencial em  $m$ . A rede da Figura 6.5 ilustra um pior caso. O valor do fluxo máximo é  $2L$ . Se os caminhos aumentantes  $s, a, b, t$  e  $s, b, a, t$  forem alternadamente escolhidos pelo algoritmo, então o valor do incremento de fluxo  $F'$  é sempre igual a 1 e são necessárias  $2L$  iterações. Observe que  $L$  pode ser arbitrariamente grande.

## 6.5 Um Algoritmo $O(nm^2)$

Dada uma rede  $D$  e um fluxo  $f$  em  $D$ , o Algoritmo 6.1 da seção anterior iterativamente constrói a rede residual  $D'(f)$  e procura um caminho aumentante, o qual se existir é utilizado para incrementar o valor do fluxo. Não há restrição à escolha do caminho aumentante, isto é, qualquer um pode ser considerado.

Na presente seção descreve-se um critério de escolha de tais caminhos, devido a Edmonds e Karp. Este critério, quando incorporado ao Algoritmo 6.1 reduz sensivelmente a sua complexidade.

O critério adotado é o seguinte:

“Escolher sempre o caminho aumentante de menor comprimento, isto é, com menor número de arestas.”

A justificativa da adoção dessa estratégia é o lema seguinte.

### Lema 6.3

Seja  $f_1$  um fluxo em uma rede  $D$  e  $k$  o comprimento do menor caminho aumentante  $p_1$  para  $f_1$ . Seja  $f_2$  o fluxo obtido em  $D$  pelo aumento de  $f_1$  através de  $p_1$ , conforme o Lema 6.2. Então o comprimento do menor caminho aumentante  $p_2$  para  $f_2$ , se houver, é  $\geq k$ .

**Prova** Comparando as redes residuais  $D'(f_1)$  e  $D'(f_2)$  verifica-se que (i) arestas não pertencentes a  $p_1$  aparecem igualmente nas duas redes, (ii) arestas de  $p_1$  com capacidade mínima em  $p_1$  não figuram em  $D'(f_2)$  e (iii) arestas diretas  $(v, w)$  de  $p_1$  com  $f_1(v, w) = 0$  implicam a introdução de arestas contrárias  $(w, v)$  em  $D'(f_2)$ . Portanto, as únicas arestas em  $D'(f_2)$  mas não em  $D'(f_1)$  são as de (iii). Se  $p_2$  possui comprimento  $< k$ , então necessariamente utiliza arestas de (iii), o que contradiz  $p_1$  ser de comprimento  $k$  (isto é, se  $(v, w)$  é parte do menor caminho  $p_1$ , a introdução em  $D$  de arestas do tipo  $(w, v)$  não pode produzir caminhos de comprimento menor do que o de  $p_1$ ). ■

Ou seja, a única modificação proposta em relação ao Algoritmo 6.1 é a forma de obter o caminho aumentante. Na descrição da seção anterior este é arbitrário, enquanto a nova estratégia o define como o de menor comprimento. Pelo Lema 6.3, os comprimentos dos caminhos aumentantes escolhidos segundo o critério apresentado formam uma sequência não decrescente. Para cada  $k$ ,  $1 \leq k < n$ , existem não mais do que  $O(m)$  caminhos aumentantes. Pois em cada tal caminho  $p$ , pelo menos uma aresta de  $p$  pode ser saturada (a de menor capacidade

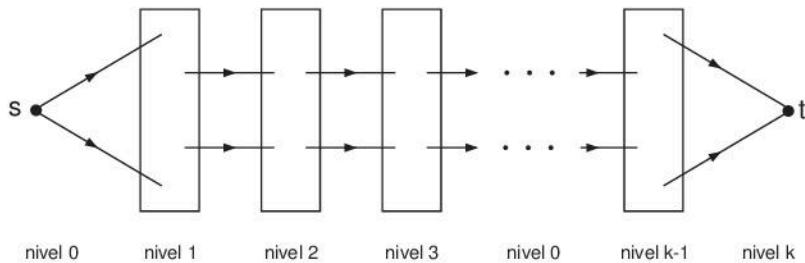


Figura 6.6: Esquema de uma rede de camadas

corrente), o que impossibilita a sua utilização posterior em outro caminho do mesmo comprimento  $k$ . Cada menor caminho aumentante pode ser encontrado em  $O(m)$  passos, utilizando busca em largura. Logo a complexidade de todo o algoritmo é  $O(nm^2)$ . Observe que essa alteração simples introduzida no Algoritmo 6.1 é suficiente para transformá-lo em um processo polinomial.

A argumentação anterior mostra que o algoritmo termina em  $O(nm^2)$  passos. Não foi utilizado o fato de que as capacidades das arestas são números inteiros. Isto significa que o processo pode ser aplicado mesmo se esta condição não for satisfeita. Ou seja, no algoritmo da presente seção, bem como nos subsequentes, admitem-se capacidades arbitrárias não negativas.

## 6.6 Um Algoritmo $O(n^2m)$

Seja  $f$  um fluxo em uma rede  $D(V, E)$ . O algoritmo da seção anterior iterativamente procura o caminho aumentante para  $f$  com menor comprimento. Isto é, o caminho  $p$  da origem  $s$  ao destino  $t$  com menor número de arestas na rede residual  $D'(f)$ .

Para melhor examinar o problema de encontrar tal caminho pode-se considerar a subrede  $D^*(f)$  de  $D'(f)$ , a qual contém somente os vértices e arestas de  $D'(f)$  que podem figurar em  $p$ . Por exemplo, um vértice cuja distância a  $s$  é maior do que a de  $s$  a  $t$  obviamente não pode pertencer a  $p$ . Da mesma forma, se  $v_1$  e  $v_2$  são vértices tais que a distância de  $s$  a  $v_1$  é menor ou igual à de  $s$  a  $v_2$  então uma aresta  $(v_2, v_1)$  também não aparece em  $p$ . Como consequência, a rede  $D^*(f)$  é acíclica e somente contém arestas entre vértices localizados em níveis contíguos na árvore de largura  $T$  de raiz  $s$ , conforme indica o esquema da Figura 6.6. Por extensão, o nível de um vértice  $v$  na árvore  $T$  é chamado *nível* de  $v$  na rede  $D^*(f)$ . Por sua vez,  $D^*(f)$  é denominado *rede de camadas* para  $f$ .

O seguinte algoritmo constrói uma rede de camadas  $D^*(f)$ , a partir da rede residual  $D'(f)$ .

---

**Algoritmo 6.2:** Construção da rede de camadas  $D^*(f)$

**Dados:** rede residual  $D'(f)$

efetuar uma busca em largura  $B$  em  $D'$  de raiz  $s$

$T :=$  árvore de largura obtida por  $B$

**para** cada aresta  $(v, w)$  de  $D'$  **efetuar**

**se**  $\text{nível}(v) \geq \text{nível}(w)$  em  $T$  **então**

        eliminar  $(v, w)$

    seja  $v_1, \dots, v_n$  a sequência de vértices de  $D'$  em ordem não crescente de seus níveis em  $T$

**para**  $j = 1, \dots, n$  **efetuar**

**se**  $v_j \neq t$  e grau saída  $(v_j) = 0$  **então**

        eliminar  $v_j$

---

O Algoritmo 6.2 possui complexidade  $O(m)$ . A sua correção decorre do fato de que numa busca em largura não existe aresta  $(v, w)$  tal que  $\text{nível}(v) < \text{nível}(w) - 1$ . Como exemplo, na Figura 6.7 se encontra a rede de camadas do fluxo da Figura 6.1(a).



Figura 6.7: Rede de camadas para o fluxo da Figura 6.1(a)

Observe que se  $f$  for máximo, não há caminho em  $D'$  de  $s$  a  $t$ . Consequentemente  $t$  não será incluído em  $D^*(f)$ . Logo, o digrafo de camadas  $D^*(f)$  não existirá. Pois todos os seus vértices serão eliminados no processo.

O algoritmo de determinação do fluxo máximo pode ser reformulado em termos do digrafo de camadas. Naturalmente, a procura de um caminho aumentante  $p$  pode ser realizada na rede de camadas  $D^*$  ao invés da rede residual  $D'$ . Suponha que foi encontrado um caminho  $p$ , da origem ao destino em  $D^*(f)$  e seja  $F'$  a capacidade mínima entre as arestas de  $p$ . A estratégia inicial consistia em aumentar de  $F'$  o fluxo  $f$  em  $D$ , obtendo o fluxo aumentado  $f'$ , o qual seria utilizado para construir  $D^*(f')$ , repetindo-se o processo. Como alternativa, tenta-se obter  $D^*(f')$  diretamente de  $D^*(f)$ . Para tanto, basta percorrer  $p$  em  $D^*(f)$ , redefinindo as capacidades  $c'(e)$  das arestas. Isto é, para cada aresta  $e$  de  $p$ , efetuar

*se  $e$  é aresta direta então*

$$c'(e) := c'(e) - F'$$

*se  $c'(e)$  decresceu para 0 então*

*eliminar  $e$  de  $D^*(f)$*

*caso contrário (quando  $e$  é aresta contrária)*

$$c'(e) := c'(e) + F'$$

O procedimento anterior atualiza a rede de camadas  $D^*(f)$  em tempo  $O(k)$ , onde  $k < n$  é a distância de  $s$  a  $t$  em  $D^*(f)$ . Contudo, este método somente pode gerar redes  $D^*(f')$  caso a distância entre  $s$  e  $t$  em  $D^*(f')$  se mantenha igual a  $k$ . Isto é, se  $p$  foi o último caminho aumentante do comprimento  $k$ , o procedimento não pode ser aplicado, pois geraria uma rede de camadas desconexa. Nesse caso, obtém-se  $D^*(f')$  a partir da definição, utilizando o Algoritmo 6.2.

Observe que o número de vezes em que a rede de camadas  $D^*$  deve ser construída pelo Algoritmo 6.2 é  $O(n)$ , isto é, no máximo uma vez para cada comprimento  $k$ ,  $1 \leq k \leq n - 1$ . Nas demais vezes obtém-se  $D^*$  pelo procedimento apresentado, mais eficiente do que a aplicação do Algoritmo 6.2. Conforme mencionado, o novo método pode ser empregado sempre que houver, na rede de camadas atualizada  $D^*(f')$ , um caminho aumentante do mesmo comprimento  $k$  obtido na rede anterior  $D^*(f)$ . Isto é, enquanto o fluxo em  $D^*$ , definido pelos caminhos aumentantes do mesmo comprimento  $k$ , não for maximal. *Com isso pode-se transformar o problema de determinar um fluxo máximo em uma rede arbitrária  $D$  em  $O(n)$  problemas de determinar um fluxo maximal em uma rede de camadas  $D^*$ .* Este último é certamente mais simples. A formulação seguinte descreve a nova estratégia.

Ao final do processo  $f$  é um fluxo máximo de valor  $F$  em  $D$ . A complexidade do Algoritmo 6.3 é basicamente igual a  $n$  vezes a complexidade do processo de obter um fluxo maximal em  $D^*$ .

Considere agora o problema de determinar um fluxo maximal  $f^*$  de valor  $F^*$  na rede de camadas  $D^*$  com origem  $s$  e destino  $t$ , sendo  $k$  a distância de  $s$  a  $t$  em  $D^*$ . A capacidade de cada aresta  $e$  em  $D^*$  será denotada  $c^*(e)$ .

O processo descrito para atualização direta de  $D^*(f)$  na realidade obtém o desejado fluxo maximal. Isto é, no passo inicial define-se  $f^*(e) := 0$ , para cada aresta  $e$  de  $D^*$ . No passo geral escolhe-se um caminho arbitrário  $p$  de  $s$  a  $t$ . Seja  $F'$  a capacidade mínima entre as arestas  $p$ . Para cada aresta  $e$  de  $p$  efetua-se  $c^*(e) := c^*(e) - F'$ , eliminando  $e$  se  $c^*(e)$  decresceu para zero. Repete-se o processo. Caso não haja caminho de  $s$  a  $t$ ,  $f^*$  é maximal.

A escolha do caminho  $v_1, \dots, v_{k+1}$  de  $s = v_1$  a  $t = v_{k+1}$  pode ser feita de modo que para cada  $j$ ,  $1 \leq j \leq k$ , o vértice  $v_{j+1}$  seja o primeiro da lista  $A(v_j)$ . Como qualquer aresta divergente de  $v_j$  conduz sempre a um nível mais próximo de  $t$ , o método está correto.

Cada caminho  $p$  é obtido portanto em tempo  $O(n)$ . Como podem haver  $O(m)$  caminhos até a separação de  $s$  e  $t$ , o processo anterior encontra um fluxo maximal em  $D^*$  após  $O(nm)$  passos. Logo, a complexidade do Algoritmo 6.3 para determinar um fluxo máximo em  $D$  é  $O(n^2m)$ .

**Algoritmo 6.3:** Fluxo máximo em uma rede (Dinic)

**Dados:** rede  $D(V, E)$ , cada aresta com capacidade real positiva origem  $s \in V$  e destino  $t \in V$

$F := 0$

**para**  $e \in E$  **efetuar**

$f(e) := 0$

construir a rede de camadas  $D^*(f)$  pelo Algoritmo 6.2

**enquanto** existir  $D^*(f)$  **efetuar**

**repetir**

obter um fluxo maximal  $f^*$  de valor  $F^*$  em  $D^*$

**para** cada aresta  $e$  de  $D^*$  **efetuar**

**se**  $e$  é aresta direta de  $D$  **então**

$f(e) := f(e) + f^*(e)$

**caso contrário**

$f(e) := f(e) - f^*(e)$

$F = F + F^*$

remover  $e$  tal que  $f^*(e) = F^*$

**até que** não exista  $D^*(f)$

construir  $D^*(f)$  pelo Algoritmo 6.2

---

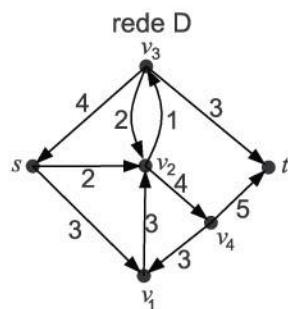


Figura 6.8: Entrada para o exemplo da Figura 6.9

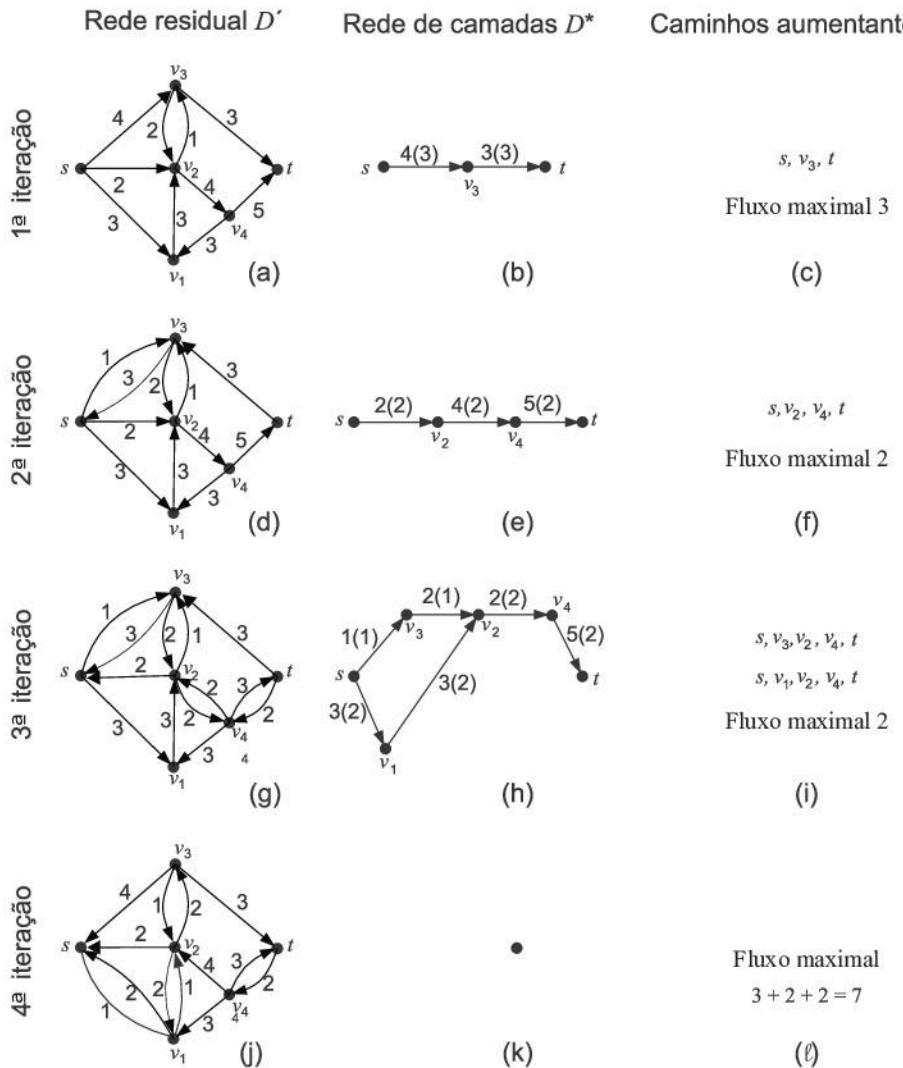


Figura 6.9: Um exemplo do Algoritmo 6.3

A Figura 6.9 ilustra um exemplo do Algoritmo 6.3, aplicado sobre a rede da Figura 6.8. O fluxo inicial é escolhido como de valor 0. A rede residual, inicial, apareceu na Figura 6.9(a), a rede de camadas correspondente na Figura 6.9(b), e o fluxo maximal, de valor 3, na Figura 6.9(c). Observe que os caminhos mínimos na rede Figura 6.9(c) possuem comprimentos 2. A rede de camadas atualizada se encontra na Figura 6.9(d), gerando a rede de camadas da Figura 6.9(e), cujos caminhos mínimos possuem comprimento 3, e o fluxo maximal com valor 2. O processo prossegue obtendo comprimento de caminho mínimo até 4, o qual aparece na Figura 6.9(i). O valor do fluxo máximo é a soma dos valores dos fluxos maximais obtidos, igual a  $3 + 2 + 2 = 7$ .

## 6.7 Um Algoritmo $O(n^3)$

Na presente seção apresenta-se um algoritmo para determinar um fluxo maximal em uma rede de camadas, devido a Malhotra, Pramodh Kumar e Maheshwari. Este algoritmo possui complexidade  $O(n^2)$ . Isto permite obter o fluxo máximo em uma rede em  $O(n^3)$  passos, conforme já mencionado.

Na seção anterior, obtinha-se um fluxo maximal eliminando-se uma aresta da rede em cada iteração. No algoritmo descrito a seguir elimina-se um vértice em cada passo.

Para cada vértice  $v$  da rede de camadas  $D^*$ , define-se *capacidade* de  $v$  como sendo  $\min\{\sum_w c^*(v, w), \sum_w c^*(w, v)\}$ , onde  $c^*(w, s) = c^*(t, w) = \infty$ . Ou seja, a capacidade de  $v$  é o menor dentre os somatórios das capacidades  $c^*$  das

arestas divergentes e convergentes a  $v$ , respectivamente. Este valor exprime o máximo de fluxo que pode passar através de  $v$ .

Seja  $v_c$  o vértice de menor capacidade em  $D^*$ . Existe um fluxo  $f_c^*$  que satura  $v_c$  em  $D^*$  e cujo valor é igual à capacidade mínima  $c^*(v_c)$ . A construção desse fluxo está indicada mais adiante. Obtido  $f_c^*$ , decrementa-se  $c^*(e)$  de  $f_c^*(e)$ , e incrementa-se  $f^*(e)$  de  $f_c^*(e)$ , para cada aresta  $e$  de  $D^*$  com  $f_c^*(e) \neq 0$ . Se  $v_c$  é igual a  $s$  ou  $t$  o processo termina, pois um fluxo que satura a origem ou o destino é obviamente maximal. Considere então  $v_c \neq s, t$ . Elimina-se  $v_c$  de  $D^*$ , pois esse vértice está saturado. Da mesma forma, é eliminada de  $D^*$  cada aresta cuja capacidade tornou-se nula. Em seguida são eliminados todos os vértices

- (i)  $w \neq t$  com grau saída( $w$ ) = 0 e
- (ii)  $w \neq s$  com grau entrada( $w$ ) = 0.

Repetir esse passo de eliminação até que os graus de entrada e saída de cada vértice  $\neq s, t$  sejam ambos diferentes de zero. Se  $s$  ou  $t$  foi eliminado o algoritmo também se encerra pois o fluxo é maximal. Caso contrário, repetir todo o processo escolhendo o novo vértice  $v_c$  de capacidade mínima na rede corrente  $D^*$  e assim por diante.

Em seguida, descrevemos um método para obter o fluxo  $f_c^*$  de valor igual a  $c^*(v_c)$  e que sature  $v_c$ . Considere inicialmente o trecho de  $v_c$  a  $t$ . O fluxo  $f_c^*$  é enviado de  $v_c$  até  $t$  obedecendo ao seguinte critério:

*“Para cada vértice  $v$  de  $D^*$  permite-se no máximo uma aresta divergente  $(v, w)$  satisfazendo  $0 < f_c^*(v, w) < c^*(v, w)$ ”.*

Ou seja, cada aresta considerada deve ser utilizada até a sua capacidade máxima, sempre que possível. Isto implica existir no máximo uma aresta parcialmente saturada divergente de cada vértice.

Seja  $F'_c(v)$  o valor total do fluxo convergente ao vértice  $v$ . No passo inicial,  $F'_c(v_c) = c^*(v_c)$  e  $F'_c(v) = 0$  se  $v \neq v_c$ . O processo termina quando  $F'_c(v_t)$  alcançar o valor  $c^*(v_c)$  e  $F'_c(v) = 0$  se  $v \neq v_t$ .

O fluxo de saída do vértice  $v$  é definido da seguinte maneira. Seja  $(v, w)$  a primeira aresta divergente de  $v$ , na ordem considerada. Isto é,  $w$  é o primeiro vértice de  $A(v)$ . Logo

se  $F'_c(v) \leq c^*(v, w)$  então  $f'_c(v, w) := F'_c(v)$   
caso contrário  $f'_c(v, w) := c^*(v, w)$   
 $F'_c(v) := F'_c(v) - c'(v, w)$   
 $w :=$  vértice sucessor de  $w$  em  $A(v)$   
repetir o processo

Para o processo apresentado de definição do fluxo de saída de cada  $v$ , os vértices devem ser considerados em uma ordem topológica. Ou seja, nenhum vértice pode ser considerado sem que os seus antecessores na rede em  $D^*$  o tenham sido.

Para o trecho de  $s$  a  $v_c$  o método é similar. Exceto que o fluxo se desenvolve da origem  $v_c$  ao destino  $s$  na rede simétrica de  $D^*$  (isto é, invertendo-se a direção de cada aresta de  $D^*$ ).

Para a determinação da complexidade do algoritmo observe que para cada vértice  $v$ , se  $q$  arestas  $(v, w)$  foram manipuladas no processo, então pelo menos  $q - 1$  foram eliminadas. Se o custo da eliminação das arestas for contabilizado em separado, cada vértice  $v$  manipulou no máximo uma aresta. Assim sendo, a determinação de cada  $f_c^*$  é realizada em  $O(n)$  passos. Esse procedimento deve ser repetido no máximo uma vez para cada vértice. Logo a complexidade é  $O(n^2)$  mais o custo da eliminação das arestas. Cada aresta foi eliminada uma vez no máximo. Logo o custo total de eliminação é  $O(m)$ , o que mantém  $O(n^2)$  como a complexidade do algoritmo de determinação do fluxo maximal em  $D^*$ . Ou seja, um algoritmo de complexidade  $O(n^3)$  para encontrar o fluxo máximo em  $D$ .

Como exemplo, considere a obtenção de um fluxo maximal na rede de camadas  $D^*$  da Figura 6.10(a). Os valores das capacidades dos vértices estão indicados na Figura 6.10(b). O vértice  $v_1$  possui capacidade mínima 2. Um fluxo de valor 2 é enviado de  $v_1$  até  $t$  em  $D^*$  e outro de mesmo valor de  $v_1$  até  $s$  na rede simétrica de  $D^*$ . Compondo o primeiro com o simétrico do segundo obtém-se um fluxo de valor 2 de  $s$  até  $t$ , o qual satura  $v_1$ . Atualizam-se os valores das capacidades das arestas e eliminam-se  $v_1$  e os demais vértices que se tornaram irrelevantes na rede. A nova situação está indicada na Figura 6.10(c), de onde se inicia uma nova iteração e assim por diante. O fluxo maximal obtido é o da Figura 6.10(g).

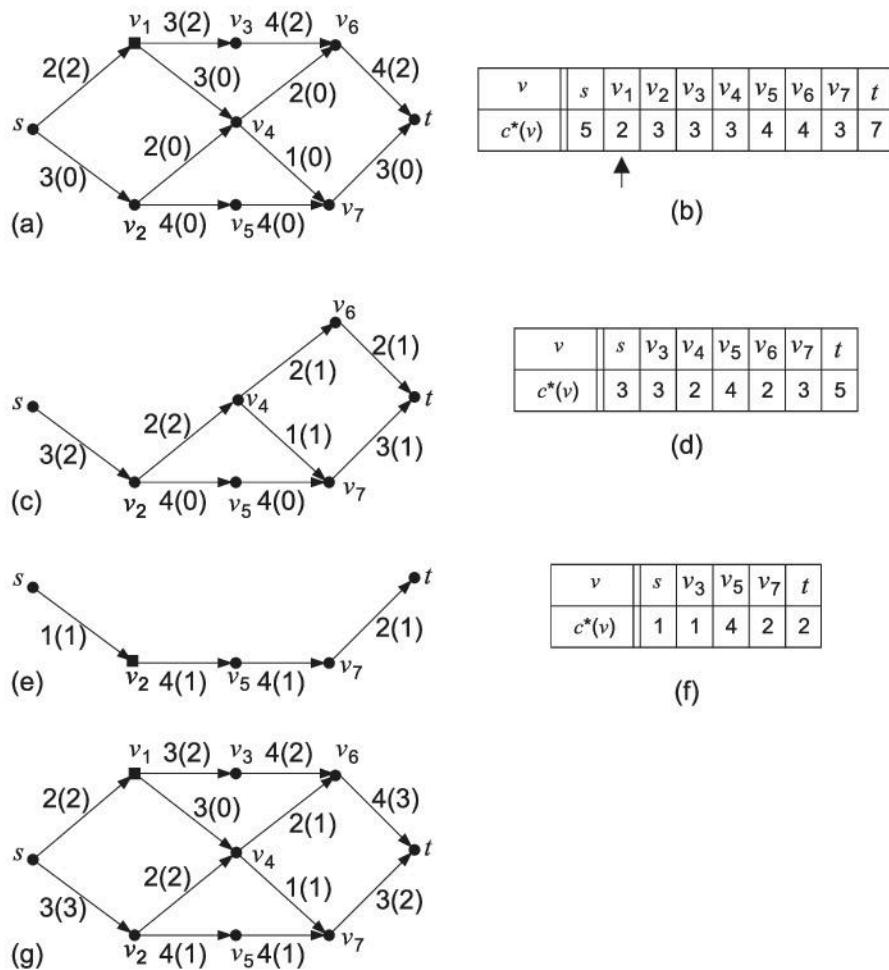


Figura 6.10: Um exemplo do algoritmo de fluxo maximal

## 6.8 Programas em Python

Esta seção contém implementações dos algoritmos formulados neste capítulo. As implementações seguem as descrições gerais apresentadas nos Capítulos 1 e 2.

### 6.8.1 Algoritmo 6.1: Fluxo Máximo

A implementação do Algoritmo 6.1 é direta. É importante notar que cada elemento do caminho retornado pela busca é um nó da lista encadeada na representação por lista de adjacências. Portanto,  $P[j].e.direta$ , por exemplo, denota o atributo `direta` do objeto aresta associado ao nó  $P[j]$ . Além disso,  $P[j].e.eD.f$  é a forma de acessar o fluxo da aresta da rede  $D$  associada a aresta  $P[j]$  retornada na busca da rede residual  $D'$ .

Programa 6.1: Fluxo máximo em uma rede (Ford e Fulkerson)

```

1 #Algoritmo 6.1: Fluxo máximo em uma rede (Ford e Fulkerson)
2 #Dados: Digrafo D com rótulo c (capacidade) nas arestas e vértices s=1 e t=D.n
3 def FluxoMaximo(D):
4     F = 0
5     for (v, uv) in D.E():
6         uv.e.f = 0
7     Dlin = ObterRedeResidual(D)
8     s, t = 1, D.n
9     P = Busca(Dlin, s, t)
10    while len(P)>0:
11        Flin = min([uv.e.r for uv in P])
12        for j in range(len(P)):
13            if P[j].e.direta:
14                P[j].e.eD.f = P[j].e.eD.f + Flin
15            else:
16                P[j].e.eD.f = P[j].e.eD.f - Flin
17        F = F + Flin
18        Dlin = ObterRedeResidual(D)
19        P = Busca(Dlin, s, t)
20    return F
21
22 def ObterRedeResidual(D):
23     """ Rede D: Digrafo com rótulo c (capacidade), f (fluxo) nas arestas, vértices s=1 e t=D.n.
24     Retorna digrafo residual, com rótulos r (resíduo), direta (direta ou não), eD (aresta ↓
25     correspondente em D)
26     """
27     Dlin = GrafoListaAdj(orientado=True)
28     Dlin.DefinirN(D.n)
29     for (v, uv) in D.E():
30         uv = uv.e
31         if uv.c-uv.f>0:
32             e = Dlin.AdicionarAresta(uv.v1, uv.v2)
33             e.r = uv.c-uv.f
34             e.direta = True
35             e.eD = uv
36             if uv.f>0:
37                 e = Dlin.AdicionarAresta(uv.v2, uv.v1)
38                 e.r = uv.f
39                 e.direta = False
40                 e.eD = uv
41     return Dlin

```

### 6.8.2 Algoritmo 6.3: Fluxo Máximo - Rede de Camadas

A função `ObterRedeResidual` encontra-se omitida na implementação a seguir, sendo igual aquela do Algoritmo 6.3. A função `Busca` foi modificada para identificar arestas visitadas que não resultaram em caminhos para o vértice de destino e removê-las ao final da busca. Esta busca é utilizada no digrafo de camadas (tais remoções equivalem à simplificação do digrafo de camadas). A busca em largura (função `BuscaLargura`) foi estendida para também marcar o nível da árvore de largura em que cada vértice se encontra, informação necessária para a criação do digrafo de camadas. A função `FluxoMaximoRedeCamadas` é uma implementação direta do Algoritmo 6.3.

Programa 6.2: Fluxo máximo em uma rede de camadas

```

#Algoritmo 6.3: Fluxo máximo em uma rede de camadas
#Dados: rede D=(V,E), cada aresta com capacidade real positiva origem s em V e destino t em V
def FluxoMaximoRedeCamadas(D):
    s,t = 1,D.n
    F = 0
    for (v,e_no) in D.E(IterarSobreNo=True):
        e_no.e.f = 0.0
    Dlin = ObterRedeResidual(D)
    Dest = ObterRedeCamadas(Dlin,s,t)
    fest = Busca(Dest,s,t) #fest = f*
    while len(fest) > 0:
        while True:
            Fest = float("inf")
            for e_no in fest:
                if e_no.e.r < Fest:
                    Fest, emin = e_no.e.r, e_no.e
            for e_no in fest:
                e_no.e.eD.f = e_no.e.eD.f + Fest * (1 if e_no.e.direta else -1)
                e_no.e.r = e_no.e.r - Fest
            F = F + Fest
            Dest.RemoverAresta(emin)
            fest = Busca(Dest,s,t) #fest = f*
            if len(fest) == 0:
                break
        Dlin = ObterRedeResidual(D)
        Dest = ObterRedeCamadas(Dlin,s,t)
        fest = Busca(Dest,s,t) #fest = f*
    return F

#Dados: rede residual Dlin (Algoritmo 6.2)
def ObterRedeCamadas(Dlin,s,t):
    Dest = GrafoListaAdj(orientado=True)
    Dest.DefinirN(Dlin.n, VizinhancaDuplamenteLigada=True)
    BuscaLargura(Dlin,s)
    for (v,elin) in Dlin.E(IterarSobreNo=True):
        w = elin.Viz
        if Dlin.Nivel[v] < min(Dlin.Nivel[w],Dlin.Nivel[t]) and elin.e.r > 0 :
            e = Dest.AdicionarAresta(v,w)
            e.eD,e.r,e.direta = elin.e.eD,elin.e.r,elin.e.direta
    #vértices não são removidos; busca em profundidade é modificada para remover arestas que ↓
    #não obtiveram sucesso em encontrar t. Assim, a condição "Dest é vazio" deve se ↓
    #trocado por "existe caminho entre s,t"
    return Dest

def BuscaLargura(D,s):

```

```

44 D.Marcado, D.EmQ = [False]*(D.n+1), [False]*(D.n+1)
45 D.Nivel = [0]*(D.n+1)
46 Q = deque()
47 D.Marcado[s], D.EmQ[s], D.Nivel[s] = True, True, 1
48 Q.append(s)
49 while len(Q) > 0:
50     v = Q[0]
51     for w in D.N(v, "+"):
52         if not D.Marcado[w]:
53             D.Marcado[w], D.EmQ[w], D.Nivel[w] = True, True, D.Nivel[v]+1
54             Q.append(w)
55     D.EmQ[v] = False
56     v = Q.popleft()

57 def Busca(D, s, t):
58     def P(v):
59         D.Marcado[v] = True
60         if v == t:
61             return True
62         for w_no in D.N(v, "+", IterarSobreNo=True):
63             w = w_no.Viz
64             if not D.Marcado[w]:
65                 Q.append(w_no)
66                 if P(w):
67                     return True
68                 ARemover.append(w_no.e)
69                 Q.pop()
70     return False

71
72 ARemover = []
73 D.Marcado = [False]*(D.n+1)
74 Q = deque()
75 P(s)
76 for e in ARemover:
77     D.RemoverAresta(e)
78 return Q
79

```

## 6.9 Exercícios

6.1 Provar ou dar contra-exemplo.

Seja  $D$  uma rede. Em todo fluxo máximo em  $D$  não existe aresta contrária

6.2 Provar ou dar contra-exemplo.

Seja  $D$  uma rede e  $(S, \bar{S})$  um corte mínimo de  $D$ . Para todo fluxo  $f$  maximal e não máximo em  $D$ , existe  $e \in (S, \bar{S})^-$  tal que  $f(e) > 0$ .

6.3 Provar que o valor do fluxo em qualquer corte de uma rede é não negativo.

6.4 Provar ou dar contra-exemplo.

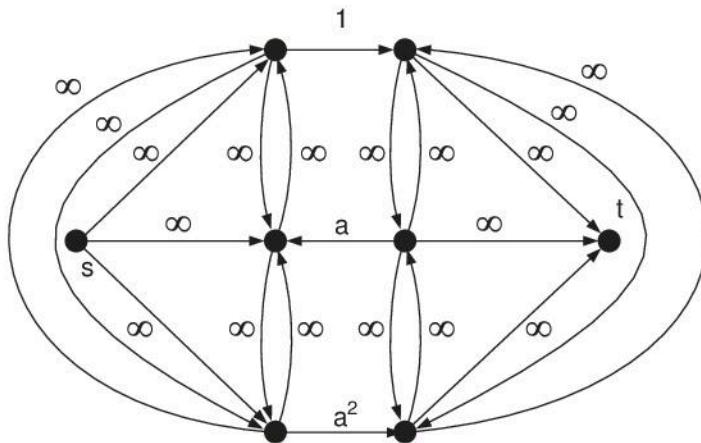


Figura 6.11: Sugestão do Exercício 6.5

Seja  $f$  um fluxo em uma rede  $D(V, E)$  com origem  $s$  e destino  $t$ . Seja  $S \subseteq V$  com  $s, t \notin S$ . Então  $f(S, \bar{S}) = 0$ .

- 6.5 Mostrar que se as capacidades das arestas de uma rede puderem assumir valores irracionais, o Algoritmo 6.1 pode levar um número infinito de passos e convergir para um valor incorreto de fluxo máximo. *Sugestão:* utilizar a rede da Figura 6.11.
- 6.6 Mostrar que se as capacidades das arestas de uma rede puderem assumir valores racionais, o Algoritmo 6.1 pode levar um número infinito de passos e convergir para um valor incorreto.
- 6.7 Formular uma implementação detalhada do algoritmo da Seção 6.5.
- 6.8 Formular uma implementação detalhada do algoritmo da Seção 6.7.
- 6.9 Seja  $D$  uma rede planar. Mostrar que é possível determinar o fluxo máximo de  $D$  em  $O(n \log n)$  passos.
- 6.10 Seja  $D$  uma rede em que as capacidades das arestas são todas unitárias. Mostrar que é possível determinar o fluxo máximo de  $D$  em  $O(m^{3/2})$  passos.
- 6.11 Seja  $G$  um grafo não direcionado. Mediante transformação em problemas de fluxo máximo, elaborar algoritmos para determinar
  - a conectividade em arestas de  $G$ .
  - a conectividade em vértices de  $G$ .
Determinar a complexidade dos algoritmos.
- 6.12 Seja  $G$  um grafo bipartido não direcionado. Um *emparelhamento* em  $G$  é um conjunto de arestas com extremidades distintas duas a duas. Um emparelhamento máximo é aquele que possui um número máximo

de arestas. Mediante transformação em um problema de fluxo, apresentar um algoritmo para encontrar um emparelhamento máximo em  $G$ .

- 6.13 Seja  $S = S_1, \dots, S_n$  uma família de subconjuntos de um conjunto. Um *sistema de representantes distintos* para  $S$  é uma sequência de elementos  $s_1, \dots, s_n$  distintos entre si e tais que  $s_i \in S_i$ ,  $1 \leq i \leq n$ . Mediante transformação em um problema de fluxo, apresentar um algoritmo para encontrar um sistema de representantes distintos. Em que condições existe a solução?
- 6.14 Seja  $D$  um digrafo. Uma *cobertura por ciclos* de  $D$  é uma coleção de ciclos simples tais que cada vértice de  $D$  pertence a exatamente um ciclo da coleção. Mediante transformação em um problema de fluxo, apresentar um algoritmo para encontrar uma cobertura por ciclos. Em que condições existe a solução?
- 6.15 Seja  $D(V, E)$  um digrafo acíclico. Um conjunto  $S \subseteq V$  é *incomparável* se para cada dois vértices distintos  $v, w \in S$ ,  $v$  não alcança  $w$  e este não alcança  $v$  em  $D$ . Um conjunto *incomparável máximo* possui um número máximo de vértices. Mediante transformação em um problema de fluxo, apresentar um algoritmo para encontrar um conjunto incomparável máximo.
- 6.16 Seja  $D(V, E)$  um digrafo acíclico. Uma *cadeia* é um caminho no fecho transitivo de  $D$ . Uma *cobertura por cadeias* é uma coleção de cadeias tal que cada vértice de  $D$  pertence a alguma cadeia da coleção. Uma cobertura *mínima* é aquela que contém o menor número de cadeias. Mediante transformação em um problema de fluxo, elaborar um algoritmo para encontrar uma cobertura mínima por cadeias.

#### 6.17 UVA Online Judge 10249

Escreva um algoritmo para o seguinte problema:

Todos os participantes da maratona ACM têm que participar do jantar de premiação. Para maximizar a interação entre os participantes, recomenda-se que, no jantar nunca se sentem à mesma mesa duas pessoas do mesmo time. Dados os números de participantes de cada time (incluindo reservas, técnicos, convidados) e o número de cadeiras de cada mesa, indicar se é possível fazer a arrumação conforme desejado e, caso seja possível, indicar a alocação às mesas.

#### 6.18 UVA Online Judge 10380

Escreva um algoritmo para o seguinte problema:

São dados os resultados parciais de um torneio de *shogi* (xadrez japonês) com  $m$  participantes. Deseja-se saber se o jogador  $p$  pode vencer o torneio e, caso isso seja possível, com qual número máximo de pontos ele pode superar o segundo colocado. Nesse torneio não há empates e, em cada jogo, um dos jogadores faz 1 ponto. Caso só seja possível para o jogador ganhar o torneio com o mesmo número de pontos de outros competidores, isso é considerado uma vitória com 0 pontos de diferença. Neste torneio cada dupla de jogadores joga duas vezes entre si.

#### 6.19 UVA Online Judge 11506

Escreva um algoritmo para o seguinte problema:

Um programador foi despedido e quer se vingar de sua empresa destruindo a conexão entre o computador de seu patrão e o principal servidor da rede. É dada a rede de interconexão entre o computador do patrão e o servidor, havendo várias máquinas intermediárias para possíveis conexões. O programador pode explodir conexões ou máquinas, menos a do patrão e o servidor. São dados os custos de explosão de cada um desses elementos e deve ser calculado o custo mínimo que o programador terá para fazer essa desconexão.

## 6.10 Notas Bibliográficas

Os trabalhos fundamentais em teoria de fluxo máximo, incluindo o Teorema 6.1 e Algoritmo 6.1, são de Ford e Fulkerson (1956, 1957 e 1962). O algoritmo da Seção 6.5 é de Edmonds e Karp (1972), o da Seção 6.6 de Dinic (1970) e o da 6.7 de Malhotra, Pramodh Kumar e Maheshwari (1978). Há outros algoritmos eficientes para o problema do fluxo máximo. Tais como: Karzanov (1974), complexidade  $O(n^3)$ . Cherkassky (1977), complexidade  $O(n^2\sqrt{m})$ . Galil (1978), complexidade  $O(n^{5/3}m^{2/3})$ . Shiloach (1978) e Galil e Naamad (1979), complexidades  $O(nm \log^2 n)$ . Sleator (1980), complexidade  $O(nm \log n)$ . Uma variação importante dos algoritmos de fluxo máximo apresentados neste capítulo foi descrita por Goldberg e Tarjan (1986), com implementação eficiente por Goldberg (1986). Uma referência indicada para o problema de fluxo máximo é o livro de Kleinberg e Tardos (2006), o qual dedica mais de uma centena de páginas a este tópico. Uma visão histórica do trabalho de Ford e Fulkerson no problema do fluxo máximo foi descrita em Schrijver (2002). Um artigo tutorial indicado de fluxo máximo é o de Goldberg, Tardos e Tarjan (1990). O Exercício 6.5 aparece em Ford e Fulkerson (1962). O exercício seguinte foi resolvido por Oliveira e Gonzaga (1983). Para o Exercício 6.9, ver Itai e Shiloach (1979) e Galil e Naamad. (1979). Os Exercícios 6.10 e 6.11 podem ser resolvidos mediante Even e Tarjan (1975). Os Exercícios 6.12 a 6.16 admitem também soluções especiais independentes do emprego de fluxo. Por exemplo, para resolver o problema do emparelhamento (Exercício 6.12) há entre outros o algoritmo de Micali e Vazirani (1980) com complexidade  $O(\sqrt{nm})$ , o qual pode ser aplicado também para grafos não bipartidos. Este assunto será tratado no Capítulo 8. As condições de existência para sistemas de representantes distintos são as de Hall (1935). O teorema básico para os tópicos dos Exercícios 6.15 e 6.16 é o de Dilworth (1950).

Página deixada intencionalmente em branco

---

# CAMINHOS MÍNIMOS

---

## 7.1 Introdução

Este capítulo é dedicado à formulação de algoritmos para resolver o problema da determinação de caminhos mínimos em grafos. Os grafos considerados são ponderados, com pesos nas suas arestas. Ao longo do capítulo, esses pesos serão denominados *distâncias*, que refletem melhor as aplicações dos algoritmos.

Serão tratadas diversas variantes dos algoritmos. Isto é, o caso em que as distâncias sejam todas não negativas, ou se admitimos também distâncias negativas. Os casos em que o objetivo seja determinar o caminho mínimo de um vértice para todos os demais, ou de todos os vértices para todos. Os casos em que o objetivo seja encontrar o  $k$ -ésimo menor caminho mínimo,  $k > 1$ , e finalmente a variação de encontrar o  $k$ -ésimo menor passeio, isto é, com possíveis repetições de vértices.

Os grafos considerados são direcionados. Com isso, admite-se que para uma certa aresta entre os vértices  $v$  e  $w$  do grafo, a distância de  $v$  para  $w$ , e de  $w$  para  $v$ , sejam diferentes entre si. Para um grafo direcionado  $D(V, E)$ , a cada aresta direcionada  $(v_i, v_j) \in E$ , será atribuída uma *distância de  $v_i$  até  $v_j$* , representada por  $w(i, j)$ . De um modo geral,  $w(i, j)$  e  $w(j, i)$  são independentes, não havendo relação entre esses valores. Os valores das distâncias geralmente são dados do problema, e arbitrários. O conjunto de todas as distâncias  $w(i, j)$  para  $v_i, v_j \in V$ , constitui uma matriz denominada *matriz de distâncias* do grafo, e representada por  $W$ .

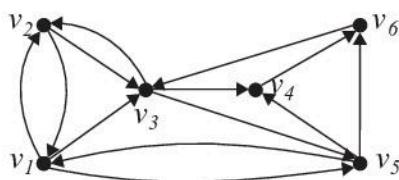
A Figura 7.1 ilustra um grafo  $D$  e sua matriz de distâncias  $W$ . Se um par  $(v_i, v_j)$  não constitui aresta do grafo, assume-se que  $w(i, j) = +\infty$ . Assim,  $w(3, 1) = +\infty$ ,  $w(1, 2) = 7$ ,  $w(2, 1) = 8$ , e assim por diante.

Para um caminho  $v_1, v_2, \dots, v_k$  dos vértices  $v_1$  ao vértice  $v_k$  do grafo  $G$ , o seu *comprimento* é definido como a soma das distâncias  $w(1, 2) + \dots, w(k - 1, k)$  das arestas contidas no caminho. O caminho mínimo de  $v_1$  para  $v_k$  é definido como aquele que possui o menor comprimento, dentre todos os caminhos de  $v_1$  para  $v_k$ , no grafo  $G$ . O *problema do caminho mínimo* é o de determinar o caminho mínimo entre os vértices desejados. O comprimento do caminho mínimo de  $v_1$  até  $v_k$  será representado por  $c(1, k)$ .

Por exemplo, no grafo da Figura 7.1, o caminho mínimo do vértice  $v_1$  para o vértice  $v_6$  é  $v_1, v_3, v_4, v_6$  de comprimento igual a  $w(1, 3) + w(3, 4) + w(4, 6) = -1 + 5 + 0 = 4$ . Isto é,  $c(1, 6) = 4$ .

Na maior parte dos algoritmos a serem apresentados, o objetivo inicial será o de determinar o comprimento do caminho mínimo. Através deste último, será possível determinar então o caminho mínimo propriamente dito.

Uma outra observação importante, diz respeito aos grafos cujos valores das distâncias possam admitir valores negativos. Mesmo para estes, consideramos sempre a restrição de que o grafo não pode conter ciclos cujo comprimento seja negativo, isto é, cuja soma das distâncias das arestas que compõem o ciclo seja negativa, pois uma vez atingido um dos vértices deste ciclo, um percurso mínimo correspondente permanecerá realizando voltas e voltas no ciclo, indefinidamente, e teria comprimento  $-\infty$ , no caso do ciclo ser de comprimento negativo.



(a)

1	0	7	-1	$\infty$	9	$\infty$
2	8	0	5	$\infty$	$\infty$	$\infty$
3	$\infty$	2	0	5	4	$\infty$
W = 4	$\infty$	$\infty$	$\infty$	3	$\infty$	0
5	9	$\infty$	$\infty$	-2	0	1
6	$\infty$	$\infty$	0	$\infty$	$\infty$	0
	1	2	3	4	5	6

(b)

Figura 7.1: Um grafo e sua matriz de distâncias

## 7.2 As Equações de Bellman

Nesta seção, desenvolveremos um método de programação dinâmica, devido a Bellman, destinado a determinar os comprimentos dos caminhos mínimos de um vértice fixo  $v_1$ , de um grafo  $D(V, E)$ , para todos os demais vértices de  $D$ , respectivamente. Este método pode ser aplicado a qualquer grafo  $D$ , inclusive aos que possuem distâncias negativas, com a ressalva observada anteriormente, da inexistência de ciclos de comprimento negativo.

O objetivo, pois, consiste em determinar os comprimentos  $c(1, k)$  dos caminhos mínimos de  $v_1$  para cada vértice  $v_k \in V$ . Em particular se  $v_k = v_1$ , a inexistência de ciclos de comprimento negativo conduz, imediatamente, a  $c(1, 1) = 0$ . Considere, agora,  $v_k \neq v_1$ . Qualquer que seja o caminho mínimo  $C$  de  $v_1$  a  $v_k$ , haverá sempre um vértice  $v_i$  que imediatamente antecede  $v_k$  em  $C$ , e, nesse caso,  $v_k$  será atingido através da aresta  $(v_i, v_k)$ . Ora, o caminho percorrido em  $C$ , desde  $v_1$  até  $v_i$  é necessariamente um caminho mínimo de  $v_1$  a  $v_i$ , caso contrário, o caminho de  $v_1$  a  $v_k$  não será mínimo. Isto nos conduz as seguintes equações de programação dinâmica, para a determinação de  $c(1, k)$

$$c(1, k) = \begin{cases} c(1, 1) = 0 & \text{se } k = 1 \\ c(1, k) = \min_{i \neq k} \{c(1, i) + w(i, k)\} & \text{se } k \neq 1 \end{cases}$$

As equações anteriores serão utilizadas como base para o desenvolvimento de algoritmos para a determinação dos caminhos mínimos, conforme será descrito nas próximas seções. Contudo elas não conduzem diretamente a algoritmos para o caso geral, pois para a sua implementação são necessárias condições adicionais.

Observando essas equações, podemos concluir que para cada vértice  $v_k \neq v_1$ , para o qual se deseja determinar o caminho mínimo partindo de  $v_1$ , é escolhido um vértice adequado  $v_i \neq v_k$ , que corresponde a aquele que antecede imediatamente a  $v_k$ , no caminho mínimo de  $v_1$  a  $v_k$ . Supondo que o vértice  $v_1$  alcance todos os demais, o conjunto dos vértices do grafo forma uma árvore direcionada enraizada  $T$ , cuja raiz é  $v_1$ , e cujo caminho de  $v_1$  a cada  $v_k$  em  $T$ , corresponde ao caminho mínimo de  $v_1$  a  $v_k$  em  $D$ . Esta árvore é então denominada *árvore de caminhos mínimos de  $v_1$* . Caso  $v_1$  não alcance todos os vértices de  $D$ , ao invés de uma árvore, teremos uma floresta de caminhos mínimos. Nesta floresta, há uma árvore com raiz  $v_1$ , composta pelos vértices alcançáveis de  $v_1$ , e as demais árvores são constituídas por raízes isoladas, formadas, respectivamente, por aqueles vértices não alcançáveis de  $v_1$ .

Como exemplo, a árvore da Figura 7.2, cuja raiz é o vértice 1, corresponde à árvore de caminhos mínimos de 1, relativa ao grafo da Figura 7.1. Com isso, obtemos  $c(1, 1) = 0$ ,  $c(1, 2) = -1 + 2 = 1$ ,  $c(1, 3) = -1$ ,  $c(1, 4) = -1 + 5 = 4$ ,  $c(1, 5) = -1 + 4 = 3$ ,  $c(1, 6) = -1 + 5 = 4$ .

## 7.3 Algoritmo de Dijkstra

Nesta seção, será apresentado o algoritmo de Dijkstra, para computar o caminho mínimo de um vértice fixo  $v_1 \in V$  do grafo direcionado  $D(V, E)$ , para todos os demais vértices de  $D$ . Supomos que todas as distâncias do grafo sejam

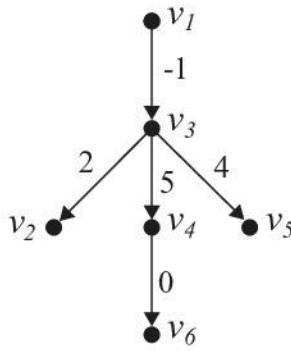


Figura 7.2: Árvore de caminhos mínimos

não negativas, isto é,  $w(i, j) \geq 0$ , para cada par  $v_i, v_j \in V$ . Caso  $(v_i, v_j) \notin E$ , por convenção,  $w(i, j) = +\infty$ . Observamos que nestas condições, não há ciclos de comprimentos negativos.

A ideia básica do algoritmo de Dijkstra é determinar um subconjunto de vértices  $V' \subseteq V$ , tal que, em cada passo, o caminho mínimo em  $D$ , de  $v_1$  até cada vértice  $v_j \in V'$  contenha apenas vértices de  $V'$ . Iterativamente, um novo vértice  $v_j$  é agregado a  $V'$ , até que se atinja  $V' = V$ , ocasião em que o algoritmo termina. Por simplicidade, o comprimento do caminho mínimo de  $v_1$  até o vértice  $v_i$  será de notado nesta seção por  $c(i)$ .

A questão a ser resolvida é então como escolher adequadamente o vértice  $v_j$  para ser agregado a  $V'$ , em cada passo. A ideia é definir cada um dos caminhos mínimos de  $v_1$  até  $v_i$  de comprimento  $c(i)$ , inicialmente, como provisório e atualizá-lo, iterativamente, até atingir um valor definitivo, que corresponde ao comprimento correto do caminho mínimo de  $v_1$  até  $v_i$ .

Assim, na iteração inicial, definir  $V' := \{v_1\}$ ,  $c(1) := 0$  e  $c(i) := w(1, i)$ . Relembreamos que  $w(1, i) = \infty$ , caso  $(v_1, v_i) \notin E$ . Nas iterações seguintes, o vértice  $v_j \in V - V'$  a ser agregado é aquele cujo valor, até então provisório, é mínimo dentre todos os comprimentos  $c(i)$ , correspondentes ao vértices  $v_i \in V - V'$ . Este valor  $c(j)$  torna-se então permanente, e o vértice  $v_j$  é adicionado a  $V'$ . Os caminhos provisórios  $c(i)$  dos demais vértices de  $V - V'$  são atualizados mediante

$$c(i) := \min\{c(i), c(j) + w(j, i)\},$$

e o processo se encerra quando  $V'$  se torna igual a  $V$ .

A formulação seguinte descreve o algoritmo de Dijkstra. São dados o grafo direcionado  $D(V, E)$ , um vértice fixo  $v_1 \in V$ , e uma distância real não negativa  $w(i, j)$ , para cada par de vértices  $v_i, v_j \in V$ . O algoritmo constrói um vetor  $c(i)$ ,  $1 \leq i \leq n$ , o qual conterá o comprimento dos caminhos mínimos de  $v_1$  para  $v_i$ .

---

**Algoritmo 7.1:** Caminhos mínimos – Dijkstra

---

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j) \geq 0$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ .

$V' := \{v_1\}$ ;  $c(1) := 0$

**para**  $v_i \in V - V'$  **efetuar**

$c(i) := w(1, i)$

**enquanto**  $V' \neq V$  **efetuar**

escolher  $v_j \in V - V'$  que minimiza o valor  $c(j)$

$V' := V' \cup \{v_j\}$

**para**  $v_i \in V - V'$  **efetuar**

$c(i) := \min\{c(i), c(j) + w(j, i)\}$

---

Um exemplo de aplicação do algoritmo aparece na Figura 7.3. O grafo dado é o da Figura 7.3(a). As distâncias entre pares adjacentes de vértices estão representadas na figura. O conteúdo do vetor  $c(i)$ ,  $1 \leq i \leq 7$ , é representado na Figura 7.3(b), com os valores correspondentes, após cada iteração. A iteração 0 correspondente ao passo inicial, onde o vértice escolhido é o dado  $v_1$ , e apenas o comprimento  $c(1) = 0$  é o permanente. Nas

demais iterações, aparece o vértice escolhido  $v_j \in V'$ , o qual minimiza  $c(j)$  em  $V - V'$ . Na iteração 1, o vértice  $v_j = v_2$  é o mínimo dentre os valores  $c(i)$ ,  $v_i \in V - V'$ , pois  $V - V' = \{v_2, v_3, v_4, v_5, v_6, v_7\}$  com  $c(2) = 1$ ,  $c(6) = 3$  e  $c(3) = c(4) = c(5) = c(7) = \infty$ . O comprimento  $c(2) = 1$  torna-se então permanente e os valores de  $c(i)$  são atualizados, para  $i = 3, 4, 5, 6, 7$ . Nesta atualização apenas o comprimento  $c(7)$  é modificado, pois  $c(1) + w(2, 7) = 1 + 2 < c(7) = \infty$ . Logo,  $c(7)$  assume o valor 3. Os demais valores de  $c(i)$  são mantidos. O conjunto  $V'$  torna-se agora  $V' = \{v_1, v_2\}$ . Na iteração 2, há dois valores de  $v_j$  que minimizam  $c(j)$  em  $V - V'$ , que são  $v_6$  e  $v_7$ , pois  $c(6) = c(7) = 3$ , e os demais valores de  $c(i)$ , para  $i = 3, 4, 5$  são todos iguais a  $\infty$ . A escolha dentre os valores de igual comprimento mínimo pode ser arbitrária. No caso, escolhemos  $v_j = v_7$  na iteração 2, e o vértice  $v_7$  é incluído em  $V'$ , que se torna  $V' = \{v_1, v_2, v_7\}$ . Ainda nesta iteração é atualizado o vetor  $c(i)$ , onde o único comprimento alterado foi  $c(4)$ , de  $\infty$  para 4, pois  $c(7) + w(7, 4) = 3 + 1 = 4 < c(4) = \infty$ . E assim por diante. Após a iteração 6, o algoritmo se encerra. Os valores finais dos comprimentos dos caminhos mínimos de  $v_i$  aparecem na última linha da tabela.

O algoritmo de Dijkstra oferece também um método simples para construir a floresta de caminhos mínimos de  $v_1$  para os demais vértices, mediante a seguinte regra, que decorre diretamente do algoritmo:

*Iniciar por uma floresta  $T(V, E_T)$ , onde  $V$  é o conjunto dos vértices de  $G$ , e  $E_T = \emptyset$ . Cada vez que um vértice  $v_j$  é escolhido, e existir algum vértice  $v_i$  que satisfaça  $c(j) + w(j, i) < c(i)$  atualizar  $E_T$  como se segue: remover a aresta de  $E_T$  que incide em  $v_i$ , caso exista, e agregar a  $E_T$  a aresta  $(v_j, v_i)$ . Ao final,  $T$  representará a floresta de caminhos mínimos desejada.*

A floresta de caminhos mínimos fornece diretamente os caminhos mínimos desejados, além dos comprimentos obtidos.

A Figura 7.3(c) ilustra a floresta de caminhos mínimos iniciadas em  $v_1$ , para o grafo de Figura 7.3(a). No caso, como  $v_1$  alcança todos os demais vértices de  $G$ , a floresta obtida é composta por uma única árvore.

A determinação da complexidade do algoritmo é imediata. A iteração inicial requer tempo  $O(n)$ . Em cada iteração seguinte, a escolha do vértice  $v_j$  minimizante pode ser realizada em tempo  $O(n)$ . Uma vez escolhido  $v_j$ , a atualização de cada  $c(i)$  requer tempo apenas constante. Isto é, cada iteração consome tempo  $O(n)$ , no total. Como são  $n - 1$  iterações, além da inicial, a complexidade final é  $O(n^2)$ . Através do uso de filas de prioridade, o algoritmo pode ser implementado em tempo  $O(m \log n)$ .

### Teorema 7.1

O Algoritmo de Dijkstra está correto.

**Prova** A prova de correção do algoritmo pode ser realizada por indução no tamanho do conjunto  $V'$ . A hipótese de indução é que, ao longo do algoritmo, para cada  $v_i \in V'$ ,  $c(i)$  é o comprimento do caminho mínimo de  $v_1$  até  $v_i$ , caminho esse contendo apenas vértices de  $V'$ .

Para a base de indução,  $|V'| = 1$ , o que implica  $V' = \{v_1\}$ , e  $c(1) = 0$ , o que está correto. Suponha que o algoritmo não esteja correto e considere a primeira iteração onde o vértice escolhido  $v_j$  é tal que  $c(j)$  não é o comprimento do caminho mínimo de  $v_1$  até  $v_j$ . Nesse caso, como os caminhos obtidos pelo algoritmo estão restritos a conter apenas vértices de  $V'$ , concluímos que o caminho de  $v_1$  até  $v_j$  que é de fato mínimo, contém um vértice  $v_k \notin V'$ . A situação é ilustrada na Figura 7.4. O caminho mínimo de  $v_1 \in V'$  até  $v_j \notin V'$  é composto pelo caminho  $C_1$ , de  $v_1$  até  $v_k$ , seguido do caminho  $C_2$  de  $v_k$  até  $v_j$ . Por hipótese, este caminho total é de comprimento menor do que o caminho  $C_3$ , de  $v_1$  até  $v_j$ , o qual está restrito a conter vértices de  $V'$ . Observamos também que o vértice que imediatamente antecede a  $v_k$ , em  $C_1$ , se encontra necessariamente em  $V'$ , caso contrário esta não seria a primeira iteração onde o caminho até o vértice escolhido não seria mínimo. Mas nesse caso, o vértice escolhido pelo algoritmo, na iteração em questão, não seria  $v_j$ , mas sim  $v_k$ . Então o algoritmo está correto. ■

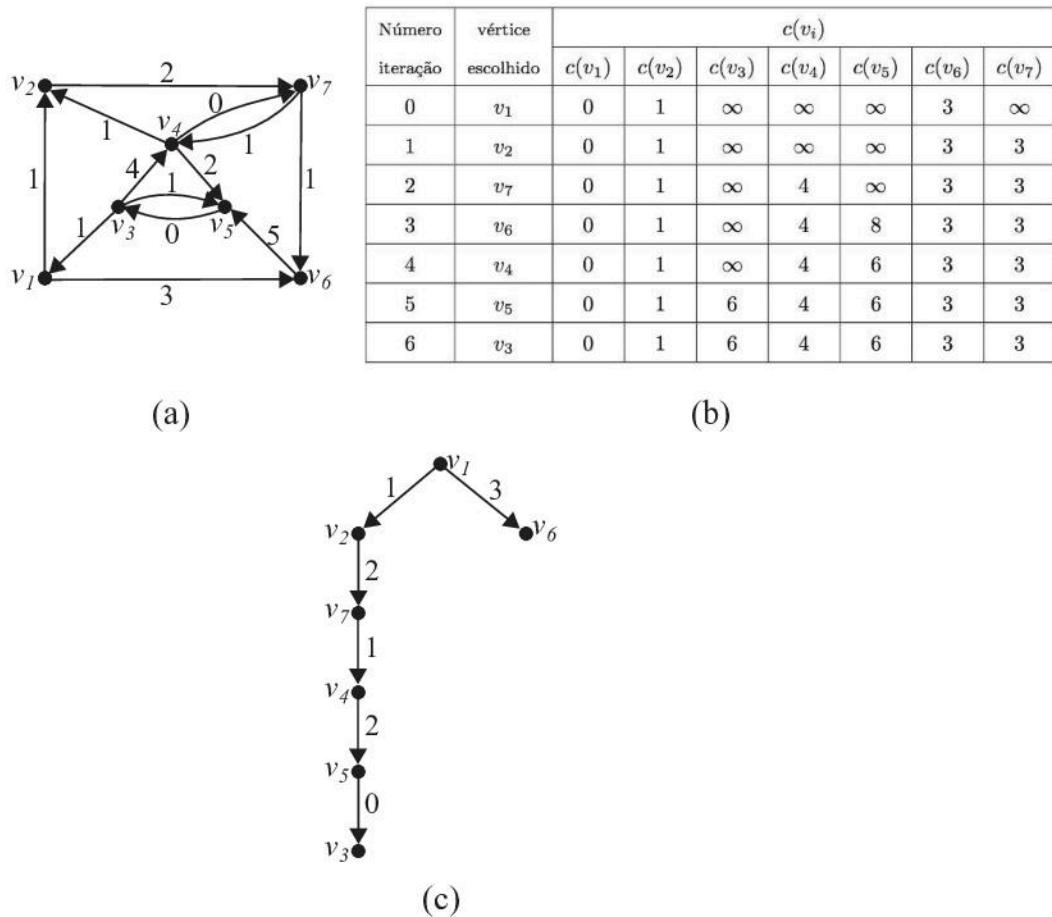


Figura 7.3: Exemplo para o algoritmo de Dijkstra

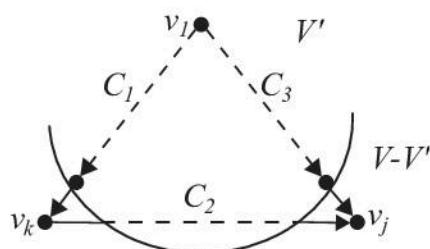


Figura 7.4: Teorema 7.1

## 7.4 O Algoritmo de Bellman-Ford

Nesta seção descreveremos um algoritmo para determinar os comprimentos dos caminhos mínimos de um dado vértice  $v_1$  para todos os demais, em um grafo direcionado  $D(V, E)$ . Cada aresta  $(v_i, v_k) \in E$  possui distância  $w(i, k)$ , onde  $w(i, k)$  é um número real. Contudo, supomos que  $D$  não contém ciclos de comprimento negativo.

O algoritmo utiliza programação dinâmica. Na realidade, ele pode ser interpretado como uma aplicação das equações de Bellman, descritas na Seção 7.2, mediante a introdução de condições adicionais. Ele se baseia em uma propriedade básica, utilizada implicitamente por todos os algoritmos de caminhos mínimos.

### Lema 7.1

Para um digrafo  $D(V, E)$ , sem ciclos negativos, e para qualquer par de vértices distintos  $v_i, v_k \in V$ , existe sempre um caminho (sem vértices repetidos) de  $v_i$  para  $v_k$  que possui comprimento mínimo, dentre todos os passeios de  $v_i$  para  $v_k$  em  $D$ .

**Prova** Se um passeio  $P$  contém algum vértice repetido  $v_i$ , o trecho entre duas ocorrências consecutivas de  $v_i$  constitui um ciclo. Remover os vértices de  $P$  entre essas ocorrências de  $v_i$ , deixando apenas a primeira delas. Repetir a operação até que não existam mais vértices repetidos. Em cada remoção foi retirado um ciclo de  $P$ . O passeio  $P$  foi então transformado em um caminho  $C$  (sem vértices repetidos). Como  $D$  não contém ciclos negativos, o comprimento de  $C$  não é maior do que o de  $P$ . ■

Como consequência do Lema 7.1, concluímos que o número de arestas de qualquer caminho mínimo de  $D$  é menor ou igual a  $n - 1$ .

Seja  $c(v_i, v_k)$  o comprimento de um caminho mínimo  $C$ , de  $v_i$  para  $v_k$ . Para a formulação das equações de programação dinâmica, introduzimos também um parâmetro  $\ell$ , que induz a quantidade máxima de arestas que  $C$  pode conter. Além disso, como tratamos de caminhos mínimos de um vértice fixo  $v_1$  para os demais, por simplicidade, omitimos a origem  $v_1$ , na notação. Assim, representamos por  $c(\ell, k)$  o comprimento do caminho mínimo de  $v_1$  para  $v_k$  em  $D$ , contendo  $\ell$  arestas, no máximo. Assim,  $c(n - 1, v_k)$  representa o comprimento do caminho mínimo de  $v_1$  para  $v_k$ , que se deseja obter, sem restrição na quantidade de arestas.

A recorrência básica na qual se baseia a programação dinâmica consiste em aplicar as equações de Bellman, da Seção 7.2, com as restrições da quantidade máxima de arestas nos caminhos, conforme a seguir.

Supondo que conhecemos o caminho mínimo  $C$ , de  $v_1$  para  $v_k$ , as seguintes alternativas podem ocorrer, em relação a  $c(\ell, k)$  para  $\ell > 0$ .

- Se  $C$  contém  $\ell - 1$  arestas, no máximo,  $\ell > 1$ , então

$$c(\ell, k) = c(\ell - 1, k). \quad (i)$$

- Se  $C$  contém  $\ell$  arestas, e  $v_i$  é o vértice que imediatamente antecede  $v_k$  em  $C$ , então

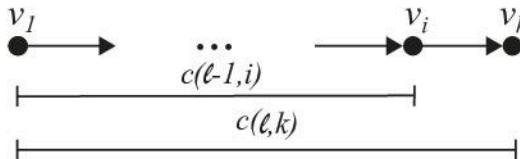
$$c(\ell, k) = c(\ell - 1, i) + w(i, k). \quad (ii)$$

Não conhecendo  $C$  porém, resta a alternativa de tentar todas as possibilidades. Isto é, (ii) se transforma em

$$c(\ell, k) = \min_{v_i \in V} \{c(\ell - 1, i) + w(i, k)\}. \quad (ii')$$

Assim, o valor de  $c(\ell, v_k)$  é o menor dentre os obtidos pelas equações (i) e (ii'). Veja a Figura 7.5.

As equações anteriores constituem a base do algoritmo de Bellman-Ford, descrito a seguir. São dados um grafo direcionado  $D(V, E)$  sem ciclos negativos, uma distância  $w(i, k)$ , para cada aresta  $(v_i, v_k) \in E$ , e um vértice fixo  $v_1 \in V$ . O algoritmo computa os comprimentos  $c(\ell, k)$ , para  $\ell = 0, \dots, n - 1$  e  $v_k \in V$ . O comprimento do caminho mínimo de  $v_1$  para  $v_k$  é então  $c(n - 1, k)$ . Esses comprimentos são armazenados em uma matriz  $C$ , de dimensão  $n \times n$ .

Figura 7.5: Cálculo de  $c(\ell, k)$ **Algoritmo 7.2:** Caminhos mínimos: Bellman-Ford

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j)$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ .

```

 $c(0, 1) := 0$ 
para  $i = 2, \dots, n$  efetuar
   $c(0, i) := \infty$ 
para  $\ell = 1, \dots, n - 1$  efetuar
  para  $k = 1, \dots, n$  efetuar
     $c(\ell, k) := \min\{c(\ell - 1, k), \min_{1 \leq i \leq n} \{c(\ell - 1, i) + w(i, k)\}\}$ 

```

Através do armazenamento dos índices  $i$  minimizantes da expressão de  $c(\ell, k)$ , é possível construir os caminhos mínimos propriamente ditos, aém de seus comprimentos.

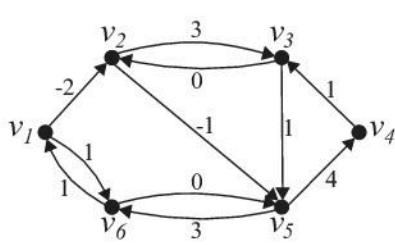
Como exemplo, seja o grafo da Figura 7.6(a), fornecido como entrada para o Algoritmo 7.2. Os números representados junto as arestas representam as respectivas distâncias entre os vértices correspondentes. Assim como anteriormente, a distância de um vértice  $v_i$  para  $v_j$  é  $\infty$ , sempre que  $(v_i, v_j) \notin E$ . O algoritmo calcula a matriz  $C$ , conforme descrito na Figura 7.6(b). Os comprimentos dos caminhos mínimos de  $v_1$  para todos os vértices de  $D$  são dados pela última linha da tabela, que corresponde aos comprimentos  $c(5, 1), \dots, c(5, 6)$ .

A determinação da complexidade é imediata. O algoritmo calcula  $c(\ell, k)$  para  $\ell = 0, 1, \dots, n - 1$  e  $k = 1, \dots, n$ . Assim são  $n^2$  valores. Para cada  $c(\ell, k)$  calculado, o algoritmo pode efetuar  $O(n)$  comparações, correspondentes a minimização  $\min_i$ . Logo, a complexidade final é  $O(n^3)$ .

Uma ligeira variação deste algoritmo pode ser implementada em complexidade  $O(nm)$ , ao invés de  $O(n^3)$ . Para tal, na minimização, ao invés de percorrer todos os valores  $1 \leq i \leq n$ , basta considerar os valores de  $i$ , tais que  $(v_i, v_k)$  é uma aresta de  $D$  (Exercício 7.9).

## 7.5 O Algoritmo de Floyd

Esta seção é dedicada a descrever o algoritmo de Floyd, o qual determina os comprimentos dos caminhos mínimos entre cada par de vértices de um digrafo  $D(V, E)$ , onde cada aresta  $(v_i, v_j) \in E$  é associada uma distância  $w(i, j)$ , representada por um número real. Por convenção, se  $(v_i, v_j) \notin E$  então  $w(i, j) = \infty$ . Os valores  $w(i, j)$  compõem

(a) Grafo  $D$ 

$\ell \backslash V$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$
0	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1	0	0	-2	$\infty$	$\infty$	1
2	0	-2	1	$\infty$	-3	1
3	0	-2	1	1	-3	0
4	0	-2	1	1	-3	0
5	0	-2	1	1	-3	0

(b) Matriz dos caminhos mínimos

Figura 7.6: Exemplo para o algoritmo de Bellman-Ford

a matriz de distâncias  $W$ , de dimensão  $n \times n$ . Embora  $D$  possa conter distâncias negativas, supomos que não há ciclos de comprimento negativo em  $D$ , caso de costume.

Utilizamos a seguinte terminologia. Se  $v_1, \dots, v_k$  é um caminho  $C$  no digrafo  $D$ , os vértices  $v_1, v_k$  são denominados *extremos* de  $C$ , enquanto que  $v_2, \dots, v_{k-1}$  são os *vértices internos* de  $C$ .

A ideia básica do algoritmo de Floyd consiste em determinar o caminho mínimo  $C$  do vértice  $v_i$  para o vértice  $v_j$ , com a restrição de que os vértices interiores a  $C$  pertencem todos a um subconjunto  $I = \{v_1, \dots, v_k\} \subseteq V$ . Partindo de  $I = \emptyset$ , a estratégia consiste em incrementar, passo a passo, o tamanho de  $I$ , até atingir o valor  $I = V$ . Nesta última condição, o caminho mínimo encontrado não possui restrição, e portanto representa o valor procurado.

Da mesma forma como nos algoritmos anteriores, o algoritmo determina os comprimentos dos caminhos mínimos, e não os caminhos propriamente ditos. Mas estes últimos podem ser encontrados a partir dos cálculos realizados pelo algoritmo.

Representamos por  $c(i, j, k)$ , o comprimento de caminho mínimo do vértice  $v_i$  ao vértice  $v_j$ , com a restrição de que seus vértices interiores contenham apenas vértices do conjunto  $\{v_1, \dots, v_k\}$ . Assim sendo,  $c(i, j, 0)$  é comprimento do caminho de  $v_i$  a  $v_j$ , restrito somente à aresta  $(v_i, v_j)$ , se existir. Na direção oposta,  $c(i, j, n)$  representa o comprimento do caminho mínimo de  $v_i$  a  $v_j$ , sem restrições, isto é, representa o valor procurado.

A estratégia descrita conduz, novamente, à equações de recorrências que formarão a base para um algoritmo de programação dinâmica. Iniciando com  $k = 0$ , os valores  $c(i, j, 0)$  correspondem exatamente à matriz de distâncias  $W$ , isto é,

$$c(i, j, 0) = w(i, j), \text{ para } 1 \leq i, j \leq n.$$

Para  $k > 0$ , há duas alternativas, para  $c(i, j, k)$ . A primeira considera caminhos mínimos sem conter o vértice  $v_k$ , isto é, cujo comprimento é  $c(i, j, k - 1)$ . Na segunda alternativa o caminho mínimo contém necessariamente o vértice  $v_k$ , e portanto é formado pela concatenação de dois subcaminhos, um de  $v_i$  para  $v_{k-1}$  e outro de  $v_k$  para  $v_j$ . Ambos devem ser mínimos, caso contrário, o caminho de  $v_i$  a  $v_j$  não seria mínimo. Além disso, os vértices interiores de ambos estão no subconjunto  $\{v_1, \dots, v_{k-1}\}$ . Nestas condições,  $c(i, j, k)$  será igual ao valor  $c(i, k, k - 1) + c(k, j, k - 1)$ .

As observações anteriores justificam a formulação das seguintes equações de recorrência.

$$c(i, j, k) = \begin{cases} w(i, j) & \text{se } k = 0. \text{ Caso contrário} \\ \min\{c(i, j, k - 1), c(i, k, k - 1) + c(k, j, k - 1)\} & , \end{cases}$$

para  $k = 1, \dots, n$ .

Partindo da matriz  $W_0 := W$ , o processo apresentado computa uma nova matriz  $W_k$ , em cada passo, a qual contém os comprimentos dos caminhos mínimos  $c(i, j, k)$ , utilizando a matriz  $W_{k-1}$ . A matriz final  $W_n$  contém o resultado do problema, isto é, os comprimentos dos caminhos mínimos desejados entre cada par de vértices  $v_i, v_j \in V$ .

Um aspecto importante deste método é que basta armazenar uma única matriz ao longo de todo o processo. No caso, a matriz  $W_k$  substitui a matriz  $W_{k-1}$  simplesmente pela atualização de alguns de seus valores. Isto é, considerando a linha  $k$  e a coluna  $k$  na matriz  $W_{k-1}$ , um elemento  $c(i, j)$  da matriz  $W_{k-1}$  terá seu valor alterado em  $W_k$ , se e somente se  $c(i, j) > c(i, k) + c(k, j)$ , em  $W_{k-1}$ . Neste caso,  $c(i, j)$  assume o valor  $c(i, k) + c(k, j)$  em  $W_k$ . Ver Figura 7.7.

---

**Algoritmo 7.3:** Caminhos mínimos: Floyd

---

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j)$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ .

**para**  $k = 1, \dots, n$  **efetuar**

**para**  $i = 1, \dots, n$  **efetuar**

**para**  $j = 1, \dots, n$  **efetuar**

$$w(i, j) := \min\{w(i, j), w(i, k) + w(k, j)\}$$

---

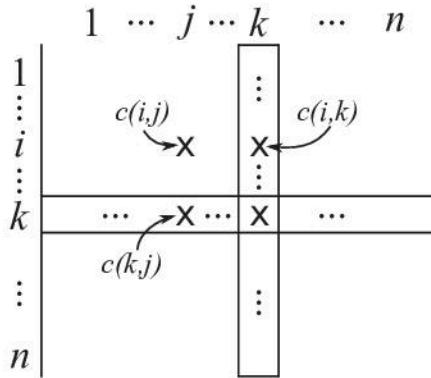


Figura 7.7: Matriz  $W_{k-1}$

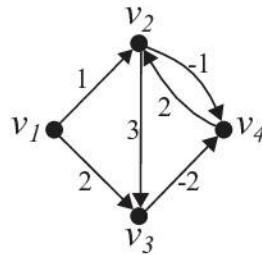


Figura 7.8: Exemplo para o algoritmo de Floyd

O Algoritmo 7.3 implementa a ideia. É dado o grafo  $G$  e a matriz de distâncias  $W$ . O algoritmo computa o comprimento do caminho mínimo  $c(i, j)$  entre cada par de vértices  $v_i, v_j$  de  $G$ . O algoritmo, possivelmente, altera a matriz  $W$ , de modo que ao final do processo o elemento  $i, j$  da matriz é o comprimento do caminho mínimo desejado.

Como exemplo, seja determinar os comprimentos dos caminhos mínimos entre cada par de vértices do grafo representado na Figura 7.8, através do algoritmo de Floyd. Para descrever a computação do algoritmo, a Figura 7.9 ilustra as matrizes  $W_k$ ,  $0 \leq k \leq n = 4$ , onde  $W_0$  é igual à matriz de distâncias,  $W$  de  $G$ , e  $W_k$  representa a matriz que contém os valores  $c(i, j, k)$ , obtida ao final de cada iteração  $k$ , do laço mais externo.

A determinação da complexidade do algoritmo é imediata. O algoritmo consiste na execução de três laços, cada qual com  $n$  iterações, onde o primeiro laço contém o segundo e este contém o terceiro. Logo, a complexidade é  $O(n^3)$ .

$$w(i, j) := w(i, k) + w(k, j)$$

do algoritmo é efetuada (Exercício 7.11).

## 7.6 $k$ -ésimos Caminhos Mínimos

Existem diversas situações em que há necessidade da determinação do segundo, terceiro, etc., de um modo geral do  $k$ -ésimo caminho de comprimento mínimo no grafo. De acordo com o Lema 7.1, existe sempre um caminho (sem vértices repetidos) cujo comprimento é mínimo, dentre todos os passeios existentes entre estes vértices. Contudo, esta situação não se aplica para  $k$ -ésimos caminhos mínimos, onde  $k > 1$ . Já para  $k = 2$ , o segundo passeio de comprimento mínimo entre os vértices especificados pode conter vértices repetidos e, nesse caso, constitui um passeio e não um caminho. Ou seja, o comprimento do segundo passeio mínimo, com vértices repetidos, pode ser

$$\begin{array}{c}
 \left| \begin{array}{cccc} 0 & 1 & 2 & \infty \\ \infty & 0 & 3 & -1 \\ \infty & \infty & 0 & -2 \\ \infty & 2 & \infty & 0 \end{array} \right| \quad \left| \begin{array}{cccc} 0 & 1 & 2 & \infty \\ \infty & 0 & 3 & -1 \\ \infty & \infty & 0 & -2 \\ \infty & 2 & \infty & 0 \end{array} \right| \quad \left| \begin{array}{cccc} 0 & 1 & 2 & 0 \\ \infty & 0 & 3 & -1 \\ \infty & \infty & 0 & -2 \\ \infty & 2 & 5 & 0 \end{array} \right| \\
 \text{(a) } W_0 \qquad \qquad \qquad \text{(b) } W_1 \qquad \qquad \qquad \text{(c) } W_2
 \end{array}$$
  

$$\begin{array}{c}
 \left| \begin{array}{cccc} 0 & 1 & 2 & 0 \\ \infty & 0 & 3 & -1 \\ \infty & \infty & 0 & -2 \\ \infty & 2 & 5 & 0 \end{array} \right| \quad \left| \begin{array}{cccc} 0 & 1 & 2 & 0 \\ \infty & 0 & 3 & -1 \\ \infty & 0 & 0 & -2 \\ \infty & 2 & 5 & 0 \end{array} \right| \\
 \text{(d) } W_3 \qquad \qquad \qquad \text{(e) } W_4
 \end{array}$$

Figura 7.9: Computação das matrizes da Figura 7.8

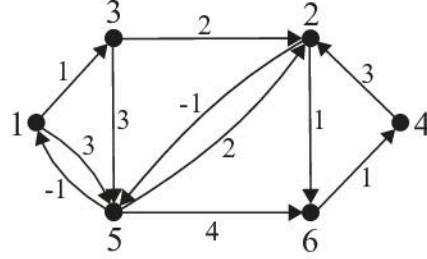


Figura 7.10: Exemplo para desvios

menor do que o comprimento do segundo caminho mínimo, onde não apareçam vértices repetidos. Nesta seção, trataremos de caminhos onde possivelmente podem aparecer vértices repetidos. Na realidade são passeios, mas para facilidade de expressão serão denominados de caminhos.

Assim, nesta seção serão apresentados algoritmos para a determinação dos  $k$ -ésimos caminhos mínimos de um grafo,  $k > 1$ , com possíveis vértices repetidos. Como sempre, o grafo será dado por sua matriz de distâncias. Estas podem ser positivas ou não. No caso de distâncias negativas, consideramos a restrição usual da inexistência de ciclos de comprimento negativos.

Representamos por  $D(V, E)$  o grafo direcionado, onde cada aresta  $(v_i, v_j)$  possui distância  $w(i, j)$ . O conjunto das distâncias constitui a matriz de distâncias  $W$ . Consideramos o problema de determinar o  $k$ -ésimo menor caminho de um vértice fixo  $v_1 \in V$ , para todos os demais  $v_i \in V$ ,  $k > 1$ . Observe que inclusive os vértices  $v_1$  e  $v_i$  podem ser repetidos.

Para simplificar o processo, estabelecemos o seguinte critério de desempate, para o caso de  $k$ -ésimos caminhos mínimos. Se  $C$  e  $C'$  são caminhos distintos e de mesmo comprimento entre dois vértices de  $D$ , aquele que seja menor lexicográfico será considerado o mínimo dos dois. Este critério se estende para qualquer  $k \geq 1$ .

O seguinte conceito é relevante para o tratamento do nosso problema de  $k$ -ésimos caminhos mínimos. Consideraremos, inicialmente o caso  $k = 2$ .

Seja  $C(i, j)$  o caminho mínimo do vértice  $v_i$  até  $v_j$ . Um *desvio* de  $C(i, j)$  relativo em um vértice  $v_p \in C(i, j)$  é um caminho  $C'$  formado pela concatenação do subcaminho de  $v_i$  a  $v_p$  em  $C(i, j)$ , seguido da aresta  $(v_p, v_q)$  não pertencente a  $C(i, j)$  e seguido do caminho mínimo de  $v_q$  a  $v_j$ .

Observamos que apesar do caminho mínimo entre dois vértices não conter vértices repetidos, um desvio deste caminho pode conter.

Como exemplo, o caminho  $C$  de comprimento 4 formado pelos vértices  $v_1, v_3, v_2, v_6, v_4$  é o caminho mínimo do vértice  $v_1$  ao vértice  $v_4$ , no digrafo da Figura 7.10. Um desvio de  $C$ , relativo à aresta  $(2, 6)$  de  $C$ , é formado pela concatenação (i) do caminho  $v_1, v_3, v_2$ , (ii) da aresta  $(v_2, v_5)$ , e (iii) do caminho  $v_5, v_2, v_4$ , o qual é mínimo de  $v_5$  a  $v_4$ , de acordo com os critérios de desempate utilizados. Assim, o desvio  $C'$  obtido é o caminho  $v_1, v_3, v_2, v_5, v_2, v_4$  de comprimento 5, onde o vértice 2 aparece repetido.

O teorema seguinte estabelece a aplicação dos desvios na determinação de  $k$ -ésimos caminhos mínimos.

### Teorema 7.2

Seja  $C(i, j)$  o caminho mínimo de um vértice  $v_i \in V$  ao vértice  $v_j \in V$ , e  $C' \neq C(i, j)$  um outro caminho de  $v_i$  a  $v_j$ . Então  $C'$  é o segundo menor caminho de  $v_i$  a  $v_j$  se e somente se  $C'$  é um desvio de  $C(i, j)$ , com o menor comprimento entre todos os desvios de  $C(i, j)$ , e  $C'$  é o menor lexicográfico dentre os caminhos de  $v_i$  a  $v_j$  que possuem o mesmo comprimento que  $C'$ .

**Prova** Por hipótese,  $C'$  é o segundo menor caminho de  $v_i$  até  $v_j$ . Seja  $C'$  formado pelos vértices  $w_1, w_2, \dots, w_\ell$ , onde  $w_1 = v_i$  e  $w_\ell = v_j$ . Seja  $w_1, \dots, w_p$  a sequência comum, a partir de  $v_1$ , entre  $C'$  e  $C(i, j)$  com comprimento máximo. Isto é,  $(w_1, w_2), (w_2, w_3), \dots, (w_{p-1}, w_p)$ ,  $p < \ell$ , são as arestas dos prefixos comuns entre  $C'$  e  $C(i, j)$ , mas  $(w_p, w_{p+1})$  é aresta de  $C'$ , mas não de  $C(i, j)$ . Examinemos o subcaminho  $C'' \subseteq C'$ , formado pelas arestas  $(w_p, w_{p+1}), \dots, (w_{\ell-1}, w_\ell)$ . Então  $C''$  necessariamente coincide com o menor caminho de  $w_{p+1}$  a  $w_\ell$ . Pois, caso contrário, o caminho formado por  $w_1, \dots, w_p, w_{p+1}$  seguido pelo caminho mínimo de  $w_{p+1}$  a  $w_\ell$  teria comprimento menor do que  $C'$ . Essa última condição, ou contraria  $C(i, j)$  como o menor caminho entre  $v_i$  e  $v_j$ , ou  $C'$  como o segundo menor caminho entre esses vértices. Então  $C''$  coincide com o segundo menor caminho de  $w_{p+1}$  a  $w_\ell$ . Isto é,  $C'$  é um desvio de  $C(i, j)$ . Como  $C''$  é o segundo menor caminho, este desvio é o de menor comprimento. A prova da recíproca é similar. ■

Para  $k$ -ésimos caminhos,  $k \geq 2$ , utilizamos a seguinte generalização do conceito apresentado.

Inicialmente, seguindo uma notação já utilizada no presente capítulo, seja

$$\begin{aligned} C_k(j) &= k\text{-ésimo caminho mínimo de } v_1 \text{ até } v_j \\ c_k(j) &= \text{comprimento de } C_k(j) \end{aligned}$$

Conforme já observado,  $C_k(j)$  é único, pois entre  $k$ -ésimos caminhos de mesmo comprimento, o menor lexicográfico será considerado o mínimo entre eles. Assim, cada  $C_k(j)$  termina por uma (única) aresta  $(i, j)$ ,  $i \neq j$ . Isto é, seja

$$\#_k(i, j) = \begin{aligned} &\text{número dos } k'\text{-ésimos caminhos de } v_1 \text{ até } v_j, \text{ que terminam} \\ &\text{na aresta } (i, j), \text{ para } k' = 1, 2, \dots, k. \end{aligned}$$

Isto é,  $\#_k(i, j)$  representa o número de vezes em que a aresta  $(i, j)$  é a última em cada um dos caminhos mínimos, do primeiro ao  $k$ -ésimo, desde  $v_1$  a  $v_j$ .

O parâmetro acima será útil na determinação de  $k$ -ésimos caminhos. Se computarmos os  $k$ -ésimos caminhos para valores crescentes de  $k$ , este parâmetro pode ser facilmente calculado mediante

$$\#_k(i, j) = \begin{cases} \#_{k-1}(i, j) + 1, & \text{se } C_k(j) \text{ termina na aresta } (i, j) \\ \#_{k-1}(i, j), & \text{caso contrário,} \end{cases}$$

definindo-se  $\#_0(i, j) = 0$ , para todo par de vértices  $v_i, v_j \in V$ .

Com esses elementos, podemos agora formular uma solução através de programação dinâmica, para o problema de encontrar  $k$ -ésimos caminhos mínimos, do vértice  $v_1$  para todos os demais, do seguinte modo.

Inicialmente, utilizando algoritmos descritos nas seções anteriores deste capítulo, determinar o caminho mínimo

$$C_1(j), \text{ para todo vértice } v_j, \text{ e seu comprimento } c_1(j)$$

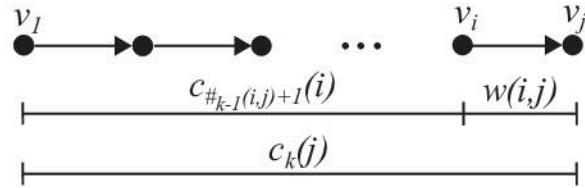


Figura 7.11: Equação (\*)

Para valores crescentes de  $k > 1$ , o comprimento  $c_k(j)$  é calculado mediante

$$c_k(j) = \min_{i \neq j} \{c_{k^*+1}(i) + w(i, j)\}, \quad (*)$$

onde  $k^* = \#_{k-1}(i, j)$

Para justificar a equação (\*) anterior suponha que  $C_k(j)$  termina na aresta  $(i, j)$ . Então  $\#_{k-1}(i, j)$  representa o número de vezes em que a aresta  $(i, j)$  foi a última no universo dos caminhos mínimos  $\{C_1(j), C_2(j), \dots, C_{k-1}(j)\}$ . Como o último caminho mínimo de  $v_1$  a  $v_j$  que utilizou esta aresta foi  $C_{k^*}(j)$ , o próximo deverá ser  $C_{k^*+1}(j)$ , o qual ainda não foi determinado. Para obter o seu comprimento, basta somar a distância  $w(i, j)$  ao comprimento  $c_{k^*+1}(i)$  do subcaminho de  $C_k(j)$ , desde  $v_1$  até  $v_i$ .

Em outras palavras, sabemos que a aresta  $(i, j)$  já foi utilizada  $k^* = \#_{k-1}(i, j)$  vezes em caminhos mínimos de  $v_1$  a  $v_j$ , já determinados. Então o subcaminho de  $v_1$  a  $v_i$  em  $C_k(j)$  também já foi utilizado  $k^*$  vezes, e todos devem ser distintos e crescentes no comprimento. Assim, para determinar  $C_k(j)$ , o subcaminho de  $v_1$  a  $v_i$  deve ter comprimento  $c_{k^*+1}(i)$ . Veja a Figura 7.11. Como não conhecemos a priori qual vértice  $v_i$  antecede  $v_j$  em  $C_k(j)$ , uma alternativa é tentar todas as possibilidades, conforme a equação (\*).

Resta ainda uma questão importante. Para que o cálculo do comprimento  $c_k(j)$  do  $k$ -ésimo caminho mínimo de  $v_1$  a  $v_j$  possa ser realizado de maneira eficiente, através da equação de programação dinâmica (\*), é necessário que todas as variáveis do lado direito da equação estejam já calculadas, antes do cálculo de  $c_k(j)$ . A computação de  $k^* = \#_{k-1}(i, j)$  é facilmente obtida, pela equação descrita. Recordamos que a computação se desenvolve para valores crescentes de  $k$ , isto é,  $k = 1, 2, \dots$ . Por hipótese,  $C_k(j)$  termina com a aresta  $(i, j)$ . Sabemos que  $k^* + 1 \leq k$ . Então se  $k^* + 1 < k$ , concluímos que  $c_{k^*+1}(i)$  certamente estará já calculado, quando do cálculo de  $c_k(j)$ , e portanto não representa um problema. Quando  $k^* + 1 = k$ , utilizaremos o critério descrito a seguir.

Para cada vértice  $v_j \in V$ , denote por  $|C_1(j)|$  o número de arestas do caminho mínimo de  $v_1$  até  $v_j$ , isto é, considerando  $k = 1$ . Denote por  $ORD_1$  a sequência de vértices de  $V$ , em ordem não decrescente dos valores  $|C_1(j)|$ . Isto é, se  $v_i$  e  $v_j$  são vértices tais que  $|C_1(i)| < |C_1(j)|$ , então  $v_i$  precede  $v_j$  em  $ORD_1$ .

O lema seguinte estabelece a ordem em que os vértices  $v_j$  devem ser calculados para a computação de  $c_k(j)$ ,  $k > 1$ .

Seja a determinação de  $c_k(j)$ , através da equação (\*),  $k > 1$ . Suponha que  $c'_k(j)$  já tenha sido calculado para todos os valores  $k' < k$  e todo  $v_j \in V$ . Seja  $(v_i, v_j)$  a última aresta em  $C_k(j)$ , e  $k^* = \#_{k-1}(i, j)$ .

### Lema 7.2

Seja  $k = k^* + 1$ . Se os comprimentos  $c_k(j)$  forem calculados segundo a ordenação dos vértices  $ORD_1$ , então os valores  $c_{k^*+1}(i)$  já terão sido determinados, quando do cálculo de  $c_k(j)$ , através da equação (\*).

**Prova** Como  $k^* = \#_{k-1}(i, j)$  e  $k = k^* + 1$ , concluímos que  $(i, j)$  é aresta final de todos os caminhos mínimos, desde o primeiro até o  $k$ -ésimo, de  $v_1$  até  $v_j$ . Como consequência, e em particular, o número de arestas  $|C_{k'}(i)|$  dos caminhos mínimos de  $v_1$  até  $v_i$  é uma unidade menor do que o número de arestas  $|C_{k'}(j)|$  dos caminhos mínimos até  $v_j$ ,  $1 \leq k' \leq k$ . Portanto, se os comprimentos dos  $k$ -ésimos caminhos mínimos  $c_k(j)$ ,  $k > 1$ , forem calculados pela equação de programação dinâmica (\*), de forma que  $v_j$  obedeca a ordenação de  $ORD_1$ , o comprimento  $c_k(i)$  já teria sido calculado, quando do cálculo de  $c_k(j)$ . ■

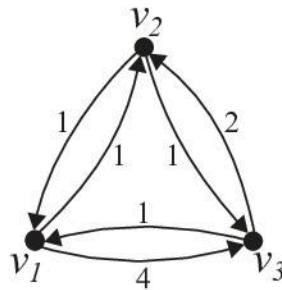


Figura 7.12: Exemplo para o Algoritmo 7.4

O algoritmo já pode ser agora formulado. A entrada é o digrafo  $D(V, E)$ , onde  $V = \{v_1, \dots, v_n\}$ , e cada par de vértices  $v_i, v_j \in V$  possui uma distância associada  $w(i, j)$ . É dado também um inteiro  $k \geq 1$ . O algoritmo a seguir encontra o  $k$ -ésimo menor caminho de  $v_1$  para cada vértice  $v_j \in V$ .

---

**Algoritmo 7.4:**  $k$ -ésimos caminhos mínimos

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j)$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ , e inteiro  $k \geq 1$ .

determinar o caminho mínimo  $C_1(j)$ , o número de arestas  $|C_1(j)|$ , e o comprimento  $c_1(j)$ , de  $v_1$  para cada  $v_j \in V$

determinar a ordenação  $ORD_1$  dos vértices  $v_j \in V$ , segundo valores não decrescentes dos números de arestas  $|C_1(j)|$ .

**para** cada aresta  $(v_i, v_j) \in E$  **efetuar**

**se**  $C_1(j)$  termina em  $(v_i, v_j)$  **então**

$\#(i, j) := 1$

**caso contrário**

$\#(i, j) := 0$

**para**  $k' = 2, 3, \dots, k$  **efetuar**

**para**  $v_j = ORD_1(1), ORD_1(2), \dots, ORD_1(n)$  **efetuar**

$\min := +\infty$

**para**  $v_i \in V - \{v_j\}$  **efetuar**

$k^* := \#(i, j)$

**se**  $c_{k^*+1}(i) + w(i, j) < \min$  **então**

$\min := c_{k^*+1}(i) + w(i, j)$

$\text{última} := (i, j)$

$c_{k'}(j) := \min$

$\#(\text{última}) := \#(\text{última}) + 1$

A complexidade do Algoritmo 7.4 é imediata. Para a determinação do caminho mínimo de  $v_1$  para cada  $v_j \in V$ , utilizamos um dos algoritmos das seções anteriores. A complexidade, portanto, é  $O(n^3)$  ou  $O(n^2)$ , dependendo se no digrafo dado há arestas de distâncias negativas, ou não. Para cada  $k' = 2, \dots, k$ , são efetuadas  $O(n^2)$  computações. Portanto, a complexidade total é  $O(kn^2)$ , além da computação de todos os caminhos mínimos do vértice origem  $v_1$  para todos os demais.

Quanto ao espaço consumido, observe que é necessário armazenar os comprimentos  $c_{k'}(j)$ , para cada  $v_j \in V$  e  $k' = 1, 2, \dots, k$ . Além disso, os valores  $\#(i, j)$ , para cada aresta  $(v_i, v_j) \in E$ . Portanto, além do digrafo  $D$  e das distâncias  $w(i, j)$ , é necessário espaço adicional para armazenar  $O(kn)$  comprimentos de caminho, e  $O(m)$  valores  $\#(i, j)$ .

Como exemplo de aplicação do Algoritmo 7.4, considere o digrafo da Figura 7.12, com os valores das distâncias assinalados nas arestas. É dado o valor  $k = 3$ , e o objetivo é determinar todos os terceiros menores caminhos de  $v_1$  para todos os vértices. Na inicialização, são determinados os seguintes parâmetros.

```
C1(1) := {v1}; |C1(1)| := 0; c1(1) := 0
C1(2) := v1, v2; |C1(2)| := 1; c1(2) := 1
C1(3) := v1, v2, v3; |C1(3)| := 2; c1(3) := 3
ORD1 := v1, v2, v3
#(1, 2) := 1; #(1, 3) := 0; #(2, 3) := 1
#(2, 1) := 0; #(3, 1) := 0; #(3, 2) := 0
```

No laço principal do algoritmo, são calculados:

$k' = 2$

```
vj = 1 ⇒ minimizante: vi = 2 ⇒ c2(1) := 2; #(2, 1) := 1
vj = 2 ⇒ minimizante: vi = 1 ⇒ c2(2) := 3; #(1, 2) := 2
vj = 3 ⇒ minimizante: vi = 1 ⇒ c2(3) := 4; #(1, 3) := 1
```

$k' = 3$

```
vj = 1 ⇒ minimizante: vi = 2 ⇒ c3(1) := 3; #(2, 1) := 2
vj = 2 ⇒ minimizante: vi = 1 ⇒ c3(2) := 4; #(1, 2) := 3
vj = 3 ⇒ minimizante: vi = 2 ⇒ c3(3) := 4; #(2, 3) := 2
```

## 7.7 $k$ -ésimos Caminhos Mínimos Simples

Nesta seção, trataremos também de algoritmos para a determinação de  $k$ -ésimos caminhos mínimos, em grafos sem ciclos negativos. Desta vez, contudo, exigindo que os caminhos sejam todos simples, isto é, sem vértices repetidos. Recordemos que no caso  $k = 1$  o caminho mínimo entre dois vértices de um grafo não contém vértices repetidos, mas a partir de  $k > 1$ , o  $k$ -ésimo caminho mínimo pode conter. Ao longo da presente seção, nos restringimos então a caminhos simples.

Similarmente à seção anterior, representamos por  $D(V, E)$  um grafo direcionado, e,  $M$ , a sua matriz de distâncias. Para um inteiro positivo  $k$ , nossa questão consiste em determinar o  $k$ -ésimo caminho simples de comprimento mínimo do vértice origem  $v_1$  ao vértice destino  $v_j$ . Representamos este caminho por  $C_k(j)$ , e seu comprimento por  $c_k(j)$ . O caminho  $C_k(j)$  é único, pois se dois caminhos possuírem comprimento idênticos, escolhemos o menor lexicográfico entre os dois, para representar o mínimo.

Para caminhos gerais, não necessariamente simples, o Teorema 7.2 estabeleceu que o segundo menor caminho entre dois vértices de  $D$  é um desvio do caminho mínimo entre eles. Estenderemos este resultado, para o caso de  $k$ -ésimo caminho mínimo simples,  $k > 1$  utilizando o conceito abaixo.

Seja  $C_k(j)$  o  $k$ -ésimo caminho mínimo simples de  $v_1$  para  $v_j$  em  $D$ . Seja  $v_p \neq v_j$ , um vértice pertencente a  $C_k(j)$ . Um *desvio simples* de  $C_k(j)$ , relativo ao vértice  $v_p \in C_k(j)$ , denominado *base* do desvio é um caminho  $\delta$  formado pela concatenação dos caminhos  $C'$  e  $C''$ , assim definidos

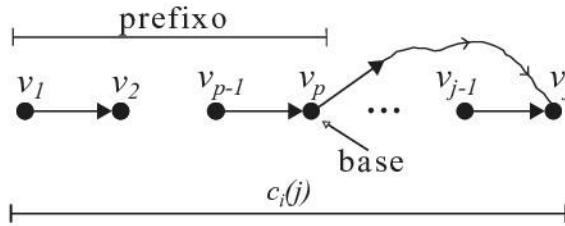
1.  $C'$  denominado *prefixo* é o subcaminho de  $v_1$  a  $v_{p-1}$  em  $C_k(j)$ .
2.  $C''$  denominado *sufixo* é o caminho mínimo de  $v_p$  a  $v_j$  com as seguintes restrições.
  - (a) Os vértices de  $C'$  não pertencem a  $C''$ .
  - (b) A aresta  $(v_p, v_{p+1})$  não pertence a  $C''$ .

Ver a Figura 7.13.

Quando  $k = 1$ , por convenção, definimos que a base de  $C_1(j)$  é o vértice  $v_1$ . O teorema seguinte ilustra a aplicação de desvios simples na determinação de  $k$ -ésimos caminhos simples.

### Teorema 7.3

Seja  $C_k(j)$  o  $k$ -ésimo caminho simples de  $v_1$  a  $v_j$ ,  $k \geq 1$ . Então o  $k$ -ésimo caminho mínimo simples de  $v_1$  a  $v_j$  é um desvio simples de  $C_{k'}(j)$ , para algum  $k'$  tal que  $1 \leq k' < k$ . Além disso,  $C_{k'}(j)$  é o desvio simples de menor comprimento entre todos os desvios de  $C_{k'}(j)$ , não utilizados nos caminhos mínimos para valores menores do que  $k$ .

Figura 7.13: Desvio simples relativo a  $C_i(j)$ 

**Prova** A prova é por indução em  $k$ . Para  $k = 1$ , não há o que provar. Para  $k > 1$ , suponha o teorema válido para qualquer inteiro até  $k - 1$ . Isto é, por hipótese, o  $i$ -ésimo caminho mínimo simples  $C_i(j)$  de  $v_1$  a  $v_j$  coincide com algum desvio simples  $C_{i'}(j)$ , para  $1 \leq i' < i < k$ . Então os desvios correspondentes também são mínimos, em ordem crescente. Seja  $\delta_i$  o desvio correspondente a  $C_i(j)$ , denote por  $\Delta_i$  o conjunto de todos os desvios de  $C_{i'}(j)$ , para todos os valores de  $i' < i$ . Então o conjunto dos  $i$ -ésimos,  $i > 1$ , caminhos mínimos  $\{C_1, \dots, C_i\}$  correspondem aos  $i - 1$  desvios mínimos contidos no conjunto  $\Delta_i$ ,  $i > 1$ . O objetivo agora é mostrar que o  $k$ -ésimo caminho mínimo simples  $C_k(j)$  é um desvio contido em  $\Delta_k$ . Observe também que os  $k - 2$  menores desvios contidos em  $\Delta_k$  já foram usados, pois correspondem aos caminhos mínimos  $\{C_2, \dots, C_{k-1}\}$ . Mais especificamente mostramos a seguir que  $C_k(j)$  coincide com o desvio mínimo simples contido em  $\Delta_k - \{C_2, \dots, C_{k-1}\}$ .

Seja  $\delta$  o desvio mínimo simples  $v_1, v_2, \dots, v_p, \dots, v_j$ , contido em  $\Delta_k - \{C_2, \dots, C_{k-1}\}$ . Então  $\delta$  é um desvio mínimo simples de algum caminho  $C_i(j)$ ,  $1 \leq i < k$ . Como  $\delta$  e  $C_k(j)$  são ambos caminhos de  $v_1$  a  $v_j$ , eles coincidem do vértice inicial  $v_1$  até um certo vértice  $v_p$ ,  $p \geq 1$ . Se  $p = j$  concluímos que  $\delta$  e  $C_k(j)$  coincidem, o que prova o teorema. Suponha então que  $p < j$ . Comparemos  $\delta$  e  $C_k(j)$ . Suponha que o comprimento de  $\delta$  seja maior do que o de  $C_k(j)$ . Considerando que todo caminho simples de  $v_1$  para  $v_j$ , corresponde a algum desvio simples de um caminho  $C_{i'}(j)$ , para algum valor de  $i'$ , a alternativa  $c(\delta) > c(C_k(j))$  implica que  $C_k(j)$  não seria o  $k$ -ésimo caminho mínimo, mas sim algum  $k$ -ésimo caminho mínimo, para  $k' < k$ , que já foi computado, uma contradição. Por outro lado, se o comprimento de  $\delta$  for maior que o de  $C_k(j)$ , implica a existência de um desvio de algum  $C_{k'}(j)$ ,  $k' < k$ , de comprimento menor do que  $\delta$ , uma outra contradição. Logo, os comprimentos são iguais. Consequentemente, os comprimentos de  $\delta$  e de  $C_k(j)$  coincidem, logo  $\delta$  e  $C_k(j)$  também coincidem.

■

A prova do Teorema 7.3 sugere um algoritmo para determinar  $k$ -ésimos caminhos, o qual, em linhas gerais é o seguinte.

Dados digrafo  $D(V, E)$ , sua matriz de distâncias, vértices  $v_1, v_j \in V$  e um inteiro  $k \geq 1$ , inicialmente determinar o caminho mínimo  $C_1(j)$ , de  $v_1$  para  $v_j$ , e definir um conjunto  $\Delta$ , com valor inicial vazio. A ideia básica consiste em determinar todos os desvios simples do caminho simples  $C_{i-1}(j)$ , de  $v_1$  para  $v_j$ , e incluí-los em  $\Delta$ . Em seguida, selecionar o menor desvio  $\delta_{\min}$  contido em  $\Delta$ , definir  $C_i(j) := \delta_{\min}$ , e removê-lo de  $\Delta$ . Este processo deve ser repetido para cada valor de  $i$ ,  $2 \leq i \leq k$ . Para que o processo esteja correto, e não ocorram repetições de desvios em  $\Delta$ , é necessário que os desvios gerados a partir de  $C_{i-1}(j)$  e incluídos em  $\Delta$ , não sejam desvios de  $C_{i'}(j)$ ,  $i' < i$ . Para alcançar este objetivo basta restringir os desvios de  $C_{i-1}(j)$  àqueles que possuam bases a partir da base de  $C_{i-1}(j)$ . Isto é, se  $v_1, \dots, v_p, \dots, v_j$  são os vértices  $C_{i-1}(j)$  e  $v_p$  é a base de  $C_{i-1}(j)$ , então os desvios a serem computados de  $C_{i-1}(j)$ , para inclusão em  $\Delta$ , devem se restringir à bases  $v_p, v_{p+1}, \dots, v_{j-1}$ , respectivamente. Aqueles desvios com bases  $v_1, \dots, v_{p-1}$  certamente já foram incluídos em  $\Delta$ , em iterações anteriores. Desta maneira, além do caminho mínimo  $C_{i-1}(j)$  é necessário conhecer o seu vértice base  $v_p$ . Além disso, certas arestas com origem  $v_p$  já podem ter sido consideradas em caminhos mínimos anteriores com o mesmo prefixo  $v_1, \dots, v_p$ . Desta forma, além de  $v_p$  é necessário conhecer quais vértices  $v' \in N^+(v_p)$  já participaram de caminhos mínimos deste mesmo prefixo. Seja  $N'(v_p) \subseteq N^+(v_p)$  o conjunto de tais vértices os quais devem ser considerados nas futuras computações. Por este motivo, cada elemento do conjunto  $\Delta$ , passa a ser constituído de uma tripla  $(\delta, v_p, N'(v_p))$ , que será utilizada para a geração de caminhos mínimos futuros. Há  $j - p$  bases possíveis  $v_p, \dots, v_{j-1}$ , a serem consideradas para os desvios de  $C_{i-1}(j)$ . Para  $q = p, p + 1, \dots, j - 1$ , construímos o grafo

$D_q$ , obtido pela remoção de certos vértices e arestas de  $D$ , do seguinte modo. O grafo  $D_q$  é construído a partir de  $D$ , removendo-se os vértices  $v_1, \dots, v_{q-1}$  e as arestas  $(v_q, v'_q)$ , para todo  $v'_q \in N'(v_q) \subseteq N(v_q)$ . O conjunto  $N'(v_q)$  é definido como:

$$N'(v_q) := \begin{cases} N'(v_p) \cup \{v_{p+1}\}, & \text{se } p = q \\ \{v_{q+1}\}, & \text{caso contrário} \end{cases}$$

O desvio relativo à base  $v_q$  é obtido concatenando-se o prefixo  $v_1, \dots, v_{q-1}$ , ao caminho mínimo de  $v_q$  a  $v_j$ , no grafo  $D_q$ . Este desvio e a base  $v_q$  são adicionados ao conjunto  $\Delta$ . Observe que se não existir caminho em  $D_q$ , entre  $v_q$  e  $v_k$ , então não há desvio relativo a  $v_q$ . Ao final da iteração  $q - j - 1$ ,  $\Delta$  contém os desvios necessários para a determinação de  $C_i(j)$ . Encontrar o desvio mínimo  $\delta_{\min}$  armazenado em  $\Delta$ , removê-lo de  $\Delta$  e definir  $C_i(j) := \delta_{\min}$ . Incrementar o valor de  $i$ , e repetir o processo.

O Algoritmo 7.5 descreve o processo.

Para determinar a complexidade do Algoritmo 7.5, observe que para cada valor de  $i = 1, \dots, k$ , são realizados, no máximo,  $j - 1$  computações de caminhos mínimos de  $v_p$  a  $v_j$ , onde  $v_p$  é a base de  $C_{i-1}(j)$  e  $j = |C_{i-1}(j)|$ . Como  $j = O(n)$ , são realizadas, no pior caso,  $O(kn^3)$  passos para o cálculo de todos os caminhos mínimos. O tamanho máximo que  $\Delta$  pode atingir é  $O(kn)$ , pois para cada valor de  $i = 2, \dots, k$  são inseridos até  $n$  triplas  $(\delta, v_p, N'(v_p))$  em  $\Delta$ , no máximo. Assim a determinação dos desvios mínimos bem como as demais operações são dominadas, em complexidade, pelo cálculo dos caminhos mínimos. A complexidade final é, então  $O(kn^3)$ .

---

**Algoritmo 7.5:**  $k$ -ésimos caminhos simples mínimos

---

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j)$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ , e inteiro  $k \geq 1$ .

$C_1(j) :=$  caminho mínimo em  $D$ , de  $v_1$  para  $v_j$

$\Delta := (C_1(j), v_1, \emptyset); i := 1$

**enquanto**  $\Delta \neq \emptyset$  e  $i \leq k$  **efetuar**

$(\delta, v_p, N'(v_p)) :=$  tripla  $\in \Delta$  cujo comprimento de  $\delta$  é mínimo

$\Delta := \Delta - \{(\delta, v_p, N'(v_p))\}$

$C_i(j) := \delta$

$v_p, v_{p+1}, \dots, v_j :=$  sufixo de  $\delta$ , onde  $v_p$  é a base

**para**  $q = p, \dots, j - 1$  **efetuar**

$D_q :=$  grafo obtido de  $D$ , pela remoção dos vértices  $v_1, \dots, v_{q-1}$  e arestas do conjunto  $\{(v_q, v'_q) | v'_q \in N'(v_q)\}$

$$N'(v_q) := \begin{cases} N'(v_p) \cup \{v_{p+1}\}, & \text{se } p = q \\ \{v_{q+1}\}, & \text{caso contrário} \end{cases}$$

**se** existir caminho de  $v_q$  a  $v_j$  no grafo  $D_q$  **então**

$C' := v_1, \dots, v_{q-1}$

$C'' :=$  caminho mínimo de  $v_q$  a  $v_j$ , em  $D_q$

$\delta := C' || C''$

$\Delta := \Delta \cup \{(\delta, v_q, N'(v_q))\}$

$i := i + 1$

**se**  $i = k$  **então**

$C_k(j)$  é o  $k$ -ésimo caminho simples mínimo de  $v_1$  a  $v_j$

**caso contrário**

$D$  não contém  $k$  caminhos simples de  $v_1$  a  $v_j$

---

Como exemplo de aplicação do Algoritmo 7.5, seja o grafo da Figura 7.12, utilizado no exemplo do Algoritmo 7.4, da seção anterior. Seja dado o valor  $k = 2$ , e o objetivo é determinar o segundo caminho mínimo simples de  $v_1$  para  $v_3$ . Portanto, devemos determinar  $C_1(3)$  e  $C_2(3)$ . O algoritmo então efetua as seguintes computações.

Inicialmente, computa o caminho mínimo  $v_1, v_2, v_3$  de  $v_1$  a  $v_3$ . Há duas iterações correspondentes ao bloco externo *enquanto*, para  $i = 1$  e  $i = 2$ . Inicialmente, obtém-se o caminho mínimo  $C_1(3)$ , e realizam-se duas iterações do bloco *para*  $q = p, p + 1, \dots, j - 1$ . Na primeira delas, é construído o caminho mínimo  $v_1, v_3$ , no grafo  $D_1$ , obtido de  $D$  pela remoção da aresta  $(1, 2)$ . Este caminho será atribuído a  $C_1(3)$  na iteração seguinte, que finaliza o processo.

## 7.8 Detecção de ciclos negativos

Ao longo do presente capítulo, nos algoritmos descritos, foi explicitamente ressaltado que os grafos considerados não poderiam conter ciclos de comprimento negativo. O motivo de tal restrição é que o caminho mínimo de um vértice  $v_1$  para  $v_j$  no grafo considerado terá comprimento  $-\infty$ , caso exista algum caminho entre  $v_1$  e  $v_j$  que passe por um ciclo negativo. Portanto, torna-se importante verificar se o grafo contém um ciclo negativo. Este é o objetivo da presente seção.

Uma outra motivação desta natureza é o fato de que embora o caminho mínimo de  $v_1$  para  $v_j$  possa ter comprimento  $-\infty$ , permanece de interesse a determinação de caminhos mínimos *simples*, isto é, que não contenham ciclos, mesmo na presença de ciclos negativos. Este caminho existe e seu comprimento é finito, desde que exista algum caminho de  $v_1$  a  $v_j$ . Contudo, a presença de ciclos de comprimento negativo no grafo faz com que a determinação do caminho simples mínimo de  $v_1$  para  $v_j$  seja um problema para o qual não são conhecidos algoritmos eficientes, de complexidade polinomial. Na realidade, este último problema é da classe NP-completo, assunto a ser abordado no Capítulo 9. Assim, torna-se importante conhecer, de antemão, se o grafo dado contém ciclos de comprimento negativo.

Consideramos o digrafo  $D(V, E)$ , em que as distâncias entre pares de vértices, sejam quaisquer, inclusive negativas.

O método mais conhecido para resolver o problema de detecção de ciclos negativos, consiste em uma extensão do algoritmo de Bellman-Ford, da Seção 7.4. Este último determina os caminhos mínimos do vértice  $v_1 \in V$  para cada  $v \in V$ , caso não existam ciclos de comprimento negativo, em  $D$ . A solução é o método de programação dinâmica descrito no Algoritmo 7.2, o qual determina o valor do comprimento mínimo  $c(\ell, k)$  do caminho de  $v_1$  até  $v_k$ , contendo no máximo  $\ell$  arestas. O algoritmo calcula esses comprimentos para  $\ell = 0, \dots, n - 1$  e  $v_k \in V$ , e o valor  $c(n - 1, k)$  é o comprimento do caminho mínimo de  $v_1$  a  $v_k$ . O objetivo da extensão do algoritmo de Bellman-Ford, a ser apresentada, é determinar o caminho mínimo de um vértice escolhido como origem para todos os demais vértices do grafo, caso este não contenha ciclos negativos. Caso contrário, se o grafo contiver algum ciclo negativo, reportar a sua existência.

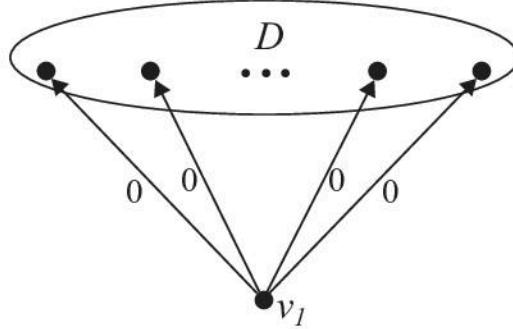
Para cumprir esse objetivo, a ideia é considerar o nosso problema nos seguintes grafos, aparentemente mais restritos. Dado o digrafo arbitrário  $D$ , definimos o digrafo  $D^+$ , denominado a *expansão* de  $D$ , adicionando um novo vértice a  $D$  rotulado como  $v_1$ , e arestas de  $v_1$  para cada vértice  $v_i \neq v_1$  de  $D^+$ . Todas as arestas  $(v_1, v_i)$ ,  $i \neq 1$ , possuem distâncias nulas. Veja Figura 7.14. Representamos por  $V$ , o conjunto de vértices de  $D^+$ , e  $n$  a sua cardinalidade  $|V|$ . A ideia consiste em resolver o problema no grafo  $D^+$ , ao invés do grafo  $D$ . Isto é possível porque os dois grafos possuem, exatamente, os mesmos ciclos, conforme o Lema 7.3.

### Lema 7.3

Seja  $D$  um digrafo e  $D^+$  a sua expansão. Então os conjuntos de ciclos de  $D$  e  $D^+$  coincidem. Em particular,  $D$  contém um ciclo negativo se e somente se  $D^+$  o contém.

**Prova**  $D$  e  $D^+$  diferem pelo vértice  $v_1$ , que pertence a  $D^+$ , mas não a  $D$ . Como  $v_1$  possui grau de entrada nulo, ele não participa de qualquer ciclo de  $D$ . Então os conjuntos de ciclos de  $D$  e  $D^+$  coincidem, em particular, aqueles que possuem comprimento negativo. ■

Para verificar se  $D$  contém ciclos negativos, utilizamos o Algoritmo 7.2, aplicado ao grafo  $D^+$ . O propósito é determinar os valores  $c(\ell, k)$ , desta vez, porém, permitindo que o parâmetro  $\ell$ , que consiste no número de arestas contidas no caminho mínimo de  $v_1$  para  $v_k$  em  $D^+$ , possa alcançar até o valor  $\ell = n$ , ao invés de estar limitado a  $\ell = 0, \dots, n - 1$ . O restante do algoritmo permanece exatamente o mesmo.

Figura 7.14: O grafo  $D^+$ 

Há um critério bastante simples para verificar se  $D^+$ , e portanto  $D$ , possui um ciclo negativo. Inicialmente, recordamos que o comprimento de um caminho mínimo de  $v_1$  para um vértice  $v_k$ , que pertença ou seja alcançável de um ciclo negativo, e cujo caminho contenha uma quantidade arbitrariamente grande de arestas, é necessariamente igual a  $-\infty$ . Tal caminho faria um número também arbitrariamente grande de percursos no ciclo, diminuindo cada vez mais o seu comprimento, até se dirigir ao vértice final  $v_k$ . Recordamos também da Seção 7.4 que, se o grafo não possui ciclos negativos, então, o número de arestas do caminho mínimo de  $v_1$  para qualquer vértice  $v_k$  é no máximo  $n - 1$ , isto é,  $c(n - 1, k)$  expressa o comprimento deste caminho mínimo. O critério para detecção de ciclos negativos é dado pelo teorema seguinte.

#### Teorema 7.4

Seja  $D$  um digrafo,  $D^+$  sua expansão,  $n$  o número de vértices de  $D^+$  e  $c(\ell, k)$  o comprimento do caminho mínimo de  $v_1$  para  $v_k$  em  $D^+$ , contendo  $\ell$  arestas, no máximo. Então  $D^+$  contém um ciclo de comprimento negativo se e somente se existir um vértice  $v_k$  de  $D^+$  tal que  $c(n, k) \neq c(n - 1, k)$ .

**Prova** Para a condição necessária, suponha que  $c(n, k) = c(n - 1, k)$ , para todo vértice  $v_k$ . Mostramos que  $D^+$  não contém ciclos negativos. Iremos verificar o que ocorre para valores maiores do que  $n$ . O valor de  $c(n + 1, k)$  pode ser calculado pela equação de programação dinâmica do Algoritmo 7.2, a partir de  $c(n, k)$ . Mas este último é igual a  $c(n - 1, k)$ , pela hipótese. Assim teremos  $c(n + 1, k) = c(n - 1, k)$ . Este argumento se aplica para qualquer  $n' \geq n$ , ou seja,  $c(n', k) = c(n' - 1, k)$ . Então  $v_k$  não pertence ou não alcança qualquer ciclo negativo do grafo. Como a hipótese se aplica a todo vértice  $v_k$ , concluímos que  $D^+$ , e portanto  $D$ , não contêm ciclos negativos.

Reciprocamente, suponha que  $D^+$  contenha um vértice  $v_k$  tal que  $c(n, k) \neq c(n - 1, k)$ , o que implica  $c(n, k) < c(n - 1, k)$ . Nesse caso, o caminho  $C$  correspondente a  $c(n, k)$  contém  $n$  arestas. Então  $C$  contém pelo menos um vértice repetido, ou seja, um ciclo  $C'$ . Suponha que  $C'$  não seja um ciclo negativo. Se o comprimento de  $C'$  for nulo, removendo  $C'$  de  $C$  obtemos um caminho de  $v_1$  a  $v_k$ , com menos do que  $n - 1$  arestas, e de comprimento menor do que  $c(n - 1, k)$ , pois o comprimento  $c(n, k)$  se mantém, um contradição. Se o comprimento de  $C'$  for positivo, a sua remoção de  $C$  produzirá um caminho de  $v_1$  a  $v_k$  de comprimento menor do que  $c(n, k)$ , também contradição. Logo,  $D^+$  necessariamente um ciclo negativo. ■

O Teorema 7.4 implica diretamente um algoritmo para verificar se um digrafo  $D$  contém ciclos negativos. Este procedimento se constitui em uma ligeira variação do algoritmo de Bellman-Ford.

Seja  $D$  um digrafo com matriz de distâncias  $W$ , definir um grafo  $D^+$  agregando a  $D$  um vértice  $v_1$  e arestas  $(v_1, v_i)$ , para cada vértice  $v_i$  de  $D$ , com distância  $w(v_1, v_i) = 0$ . A computação se desenvolve de acordo com a formulação a seguir, através da computação dos valores  $c(\ell, k)$ .

A complexidade do método apresentado é a mesma que a do algoritmo de Bellman-Ford, ou seja,  $O(n^3)$ , admitindo a implementação em  $O(nm)$  (Exercício 7.9). O Algoritmo 7.6 descreve o processo.

Uma vez verificado, através do algoritmo, que o grafo  $D$  contém algum ciclo negativo, pode ser necessário encontrar este ciclo. De acordo com o Teorema 7.4, o grafo  $D^+$  contém um vértice  $v_k$ , tal que  $c(n, v_k) \neq c(n - 1, v_k)$ . Isto significa, de acordo com a prova desse teorema, que o caminho de  $v_1$  até  $v_k$  contém um

**Algoritmo 7.6:** Detecção de ciclos negativos

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j)$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ .

$c(0, 1) := 0$

**para**  $k = 2, \dots, n$  efetuar

$c(0, k) := \infty$

**para**  $\ell = 1, \dots, n$  efetuar

**para**  $k = 1, \dots, n$  efetuar

$c(\ell, k) := \min\{c(\ell - 1, k), \min_{1 \leq i \leq n}\{c(\ell - 1, i) + w(i, k)\}\}$

**para**  $k = 1, \dots, n - 1$  efetuar

**se**  $c(n, k) \neq c(n - 1, k)$  **então**

**parar:**  $D$  contém ciclo negativo

**parar:**  $D$  não contém ciclo negativo

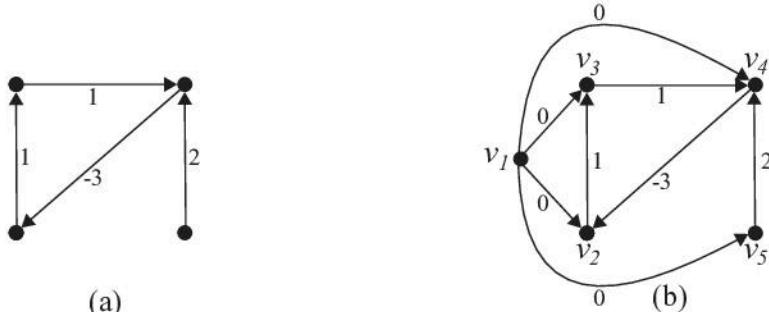


Figura 7.15: Exemplo para Algoritmo 7.6

ciclo negativo. Basta então determinar este caminho através do armazenamento dos valores de  $i$  minimizantes da expansão de  $c(\ell, k)$  do algoritmo, previamente armazenados (Exercício 7.15).

Como exemplo para o Algoritmo 7.6, seja o grafo  $D$  da Figura 7.15(a). Inicialmente, adiciona-se o vértice  $v_1$  e as arestas de  $v_1$  para os demais vértices, obtendo o grafo  $D^+$ , da Figura 7.15(b), e com os vértices rotulados, e os valores das distâncias, conforme indica a figura. Os valores de  $c(\ell, k)$  calculados pelo algoritmo encontram-se na Figura 7.15(c) e correspondem aos comprimentos dos caminhos mínimos de  $v_1$  para  $v_k$ , contendo  $\ell$  arestas no máximo, onde  $0 \leq \ell \leq 5$  e  $1 \leq k \leq 5$ . Observe que para o vértice  $v_2$ , se verifica  $c(4, 2) = -3$  e  $c(5, 2) = -6$ . Como  $n = 5$  e  $c(4, 2) \neq c(5, 2)$ , sabemos que o caminho mínimo de  $v_1$  a  $v_2$  formado por exatamente 5 arestas contém um ciclo negativo. De fato, este caminho é  $v_1, v_4, v_2, v_3, v_4, v_2$ , o qual contém o ciclo negativo  $v_4, v_2, v_3, v_4$ .

## 7.9 Programas em Python

Esta seção contém implementações dos algoritmos formulados neste capítulo. As implementações seguem as descrições gerais apresentadas nos Capítulos 1 e 2.

### 7.9.1 Algoritmo 7.1: Dijkstra

O Programa 7.1 corresponde à implementação do Algoritmo 7.1. A entrada do algoritmo é a matriz de distâncias  $w$ . A Linha 3 determina o número de vértices do digrafo, justificado pelo fato de que a primeira linha de  $w$  é sem uso. O conjunto  $V'$  é implementado como uma tabela de acesso direto  $Vlin$  tal que, para todo  $1 \leq i \leq n$ ,  $Vlin[i] = \text{True} \iff v_i \in V'$ . O bloco “enquanto  $V' \neq V$  efetuar” foi implementada na Linha 8 apenas como um contador de  $n - 1$  passos, pois este é o número exato de iterações necessárias para preencher  $Vlin$  todo com True, dado que a cada iteração uma de suas posições recebe tal valor. As Linhas 9–12 implementam o passo da escolha de  $v_j$ . Os demais passos do programa correspondem de maneira direta àqueles do algoritmo.

Programa 7.1: Caminhos mínimos - Dijkstra

```

1 #Algoritmo 7.1: Caminhos mínimos: Dijkstra
2 #Dados: matriz de distâncias w[0..n][0..n]
3 n = len(w)-1
4 Vlin = [False]*(n+1); c = [None]+[float("inf")]*n
5 Vlin[1] = True; c[1] = 0
6 for i in range(2,n+1):
7     c[i] = w[1][i]
8 for vcont in range(1,n):
9     cmin = float("inf")
10    for z in range(1,n+1):
11        if not Vlin[z] and c[z] < cmin:
12            j, cmin = z, c[z]
13    Vlin[j] = True
14    for i in range(1,n+1):
15        if not Vlin[i]:
16            c[i] = min(c[i], c[j]+w[j][i])

```

### 7.9.2 Algoritmo 7.2: Bellman–Ford

O Programa 7.2 corresponde à implementação do Algoritmo 7.2.

Programa 7.2: Caminhos mínimos - Bellman-Ford

```

1 #Algoritmo 7.2: Caminhos mínimos: Bellman-Ford
2 #Dados: matriz de distâncias w[0..n][0..n]
3 n = len(w)-1
4 c = [None]*(n)
5 for i in range(n):
6     c[i] = [None]*(n+1)
7
8 c[0][1] = 0
9 for i in range(2,n+1):
10    c[0][i] = float("inf")
11
12 for l in range(1,n):
13     for k in range(1,n+1):
14         c[l][k] = c[l-1][k]
15         for i in range(1,n+1):

```

16 |  $c[l][k] = \min(c[l][k], c[l-1][i]+w[i][k])$

A entrada do algoritmo é a matriz de distâncias  $w$ . A Linha 3 determina o número de vértices do digrafo, justificado pelo fato de que a primeira linha de  $w$  é sem uso. As Linhas 14–16 implementam o passo da atribuição de  $c(\ell, k)$ . Os passos do programa correspondem de maneira direta aqueles do algoritmo.

### 7.9.3 Algoritmo 7.3: Floyd

O Programa 7.3 corresponde à implementação direta do Algoritmo 7.3.

Programa 7.3: Caminhos mínimos - Floyd

```

1 #Algoritmo 7.3: Caminhos mínimos: Floyd
2 #Dados: matriz de distâncias w[0..n][0..n]
3 n = len(w)-1
4 for k in range(1,n+1):
5     for i in range(1,n+1):
6         for j in range(1,n+1):
7             w[i][j] = min (w[i][j], w[i][k]+w[k][j])

```

### 7.9.4 Algoritmo 7.4: $k$ -ésimo mínimo (passeio)

O Programa 7.4 corresponde à implementação do Algoritmo 7.4. As Linhas 7–9 inicializam a matriz  $c[0..k][0..n]$ , tal que  $c[i]$  corresponde ao vetor  $c_i$  de comprimentos dos  $i$ -ésimos menores caminhos de todos os demais vértices ao vértice  $v_1$ .

Na Linha 11, ocorre a chamada ao algoritmo de Dijkstra para obter o caminho mínimo de  $v_1$  a todos os demais vértices. Pressupõe-se que o retorno desta função seja um par ordenado  $(c, T)$ , onde  $c$  e  $T$  são listas tais que o valor  $c[i]$  representa o comprimento do caminho mínimo de  $v_i$  até  $v_1$  e  $T[i]$  representa o pai de  $v_i$  na árvore de caminhos mínimos associada ao vértice  $v_1$ , para todo  $1 \leq i \leq n$ . Portanto, o caminho mínimo  $C_1(i)$  de  $v_i$  até  $v_1$  é obtido, fazendo-se inicialmente  $t \leftarrow i$ , pela atualização de  $t$  com o valor de  $T[t]$  iterativamente até que  $t$  seja nulo. Neste momento, a sequência de vértices encontrados pela repetição anterior representa o caminho mínimo. As Linhas 16–18 implementam esta estratégia e aproveitam para calcular o tamanho em arestas de tal caminho, o qual será usado na determinação de  $ORD_1$ .

A implementação de  $ORD_1$  é feita como uma lista de pares  $(|C_1(i)|, i)$ , mantendo-se tais pares ordenados pela primeira coordenada. Quando um novo caminho mínimo é determinado, ele é adicionado ao final da lista e ordenado por inserção, conforme implementado pelas Linhas 19–22. Uma vez que todos os caminhos mínimos estão ordenados por número de arestas,  $ORD_1$  é redefinido pela Linha 23 para conter apenas os caminhos mínimos, mantendo a ordem final obtida, tornando-se compatível com a sua definição no algoritmo. Em seguida, define-se a matriz  $\text{Hashtag}[0..n][0..n]$ , tal que  $\text{Hashtag}[i][j]$  representa  $\#(i, j)$  no algoritmo. As Linhas 25–27 inicializam tal matriz, observando-se que  $C_1(j)$  termina em  $(v_i, v_j)$  precisamente quando  $i = T[j]$ .

Os passos restantes do programa correspondem de maneira direta àqueles do algoritmo, com a observação de que  $klin$  e  $kest$  representam respectivamente  $k'$  e  $k^*$ .

Programa 7.4:  $k$ -ésimos caminhos mínimos

```

1 #Algoritmo 7.4: k-ésimos caminhos mínimos
2 #Dados: matriz de distâncias w[0..n][0..n], natural k
3 def KesimoMinimo(w, k):
4     n = len(w)-1
5
6     #determinar o caminho mínimo C_1(j), o numero de arestas |C_1(j)| e o comprimento c_1(j) ↓
7     # de v_1 para cada v_j
8     c = [None]*(k+1)

```

```

8   for i in range(1,k+1):
9     c[i] = [0]*(n+1)
10
11 (c[1],T) = Dijkstra(w)
12
13 #determinar a ordenação ORD_1 dos vértices v_j, segundo valores não decrescentes dos ↓
14   números de arestas |C_1(j)|
15 ORD_1 = []
16 for i in range(1,n+1):
17   TamC1 = 1; t = T[i]
18   while t != None:
19     TamC1 += 1; t = T[t]
20   ORD_1.append((TamC1, i))
21 for j in range(len(ORD_1)-1, 0, -1):
22   if ORD_1[j][0] < ORD_1[j-1][0]:
23     ORD_1[j],ORD_1[j-1] = ORD_1[j-1],ORD_1[j]
24 ORD_1 = [ORD_1[i][1] for i in range(len(ORD_1))]
25
26 Hashtag = [None]*(n+1)
27 for i in range(1,n+1):
28   Hashtag[i] = [0]*(n+1)
29
30 Hashtag[1][1] = 1
31 for j in range(2, n+1):
32   Hashtag[T[j]][j] = 1
33
34 for klin in range(2,k+1):
35   for j in ORD_1:
36     minV = float("inf")
37     for i in range(1,n+1):
38       if i != j:
39         kest = Hashtag[i][j]
40         if c[kest+1][i] + w[i][j] < minV:
41           minV = c[kest+1][i] + w[i][j]
42           ultima = (i,j)
43   c[klin][j] = minV
44   Hashtag[ultima[0]][ultima[1]] = Hashtag[ultima[0]][ultima[1]] + 1
45
46 return c[k]

```

### 7.9.5 Algoritmo 7.5: $k$ -ésimo mínimo (caminho simples)

Na implementação do Algoritmo 7.5, a lista `Delta` representa o conjunto  $\Delta$  do algoritmo, porém seus elementos são quádruplas ordenadas, ao invés de triplas ordenadas, por armazenar como primeira ordenada o comprimento do caminho associado. Isto facilita o processo de obtenção do caminho de menor comprimento a cada iteração. Na implementação,  $C_i(j)$  corresponde a `C[i]`, que ao invés de armazenar apenas o caminho  $\delta$  como no algoritmo, armazena também o comprimento deste caminho. O restante do algoritmo é tradução direta para a linguagem.

Programa 7.5:  $k$ -ésimos caminhos simples mínimos

```

1 #Algoritmo 7.5: k-ésimos caminhos simples mínimos
2 #Dados: matriz de distâncias w[0..n][0..n], inteiro k >= 1, vértice inicial vi e final vj
3 def KesimoMenorSimples(w, k, vi, vj):
4   C = [None]; Delta = []
5   (c,T) = Dijkstra(w,vi)
6

```

```

7   if T[vj] != None:
8     Delta = [ (c[vj], ObterCamMinDeArv(T,vj), 1, []) ] #(comprimento, caminho mínimo, ↓
9       base, vizinhos que devem ser ignorados)
10
11    i = 1
12    while len(Delta) > 0 and i <= k:
13      (c, delta, p, Nlin) = RemoverCamMinDeDelta(Delta)
14
15      C.append((delta, c))
16      j = len(delta)
17      for q in range(p,j):
18        IgnViz = Nlin + [delta[q]] if p == q else [delta[q]]
19        wq = ObterDq(w,delta,q,IgnViz)
20        (c,T) = Dijkstra(wq,delta[q-1])
21        if T[vj] != None:
22          Clin1 = delta[:q-1]
23          Clin2 = ObterCamMinDeArv(T,vj)
24          Clin = Clin1+Clin2; cClin = sum([w[Clin[i-1]][Clin[i]] for i in ↓
25            range(1,len(Clin))])
26          Delta.append( ( cClin, Clin, q, IgnViz) )
27
28    i = i + 1
29
30  return C[k] if i > k else (None, None)
31
32 def ObterCamMinDeArv(T, v):
33   C = []
34   while v != None:
35     C.append(v)
36     v = T[v]
37   C.reverse()
38   return C
39
40 def RemoverCamMinDeDelta(Delta):
41   minV = [float("inf")]
42   for t in Delta:
43     if t[0] < minV[0]:
44       minV = t
45   Delta.remove(minV)
46   return minV
47
48 def ObterDq(w,delta,q,IgnViz):
49   n = len(w)-1
50   wq = [None]
51   for i in range(1,n+1):
52     wq.append([x for x in w[i]])
53   for i in range(q-1):
54     for j in range(1,n+1):
55       wq[delta[i]][j] = float("inf")
56   for i in IgnViz:
57     wq[delta[q-1]][i] = float("inf")
58   return wq

```

### 7.9.6 Algoritmo 7.6: Detecção de Ciclos Negativos

O Programa 7.6 corresponde à implementação direta do Algoritmo 7.6.

## Programa 7.6: Detecção de ciclos negativos

```

1 #Algoritmo 7.6: Detecção de ciclos negativos
2 #Dados: matriz de distâncias w[0..n][0..n]
3 def DeteccaoCicloNegativo(w):
4     n = len(w)-1; c = [None]*(n+1)
5     for k in range(n+1):
6         c[k] = [None]*(n+1)
7         c[0][1] = 0
8         for k in range(2,n+1):
9             c[0][k] = float("inf")
10        for l in range(1,n+1):
11            for k in range(1,n+1):
12                c[l][k] = min (c[l-1][k],
13                               min([c[l-1][i] + w[i][k]
14                                    for i in range(1,n+1)]))
15        for k in range(1,n):
16            if c[n][k] != c[n-1][k]:
17                return True
18        return False

```

## 7.10 Exercícios

- 7.1 Seja  $D(V, E)$  um digrafo sem ciclos direcionados,  $W$  sua matriz de distâncias, admitindo também valores negativos, e  $v_1 \in V$ . Formular um algoritmo para determinar os caminhos mínimos de  $v_1$  para todos os demais vértices, em complexidade linear, isto é,  $O(n + m)$ , onde  $n = |V|$ ,  $m = |E|$ .
- 7.2 Dado um digrafo  $D$  sem ciclos direcionados, formular um algoritmo para encontrar o comprimento do maior caminho em  $D$ . A matriz de distância  $W$  de  $D$  é formada por distâncias não negativas.
- 7.3 Seja  $D(V, E)$  um grafo,  $W$  sua matriz de distâncias e  $v_i, v_j \in V$  vértices arbitrários distintos de  $D$ . Um *obstáculo* de um caminho  $C$  de  $c_i$  a  $v_j$  é a aresta de distância máxima contida em  $C$ . Determinar o obstáculo de distância máxima, dentre todos os caminhos de  $v_i$  a  $v_j$  em  $D$ .
- 7.4 Considere a seguinte variação do Algoritmo 7.1, de Dijkstra, para digrafos onde a matriz de distâncias somente contém valores não negativos.

**Dados:** grafo  $D(V, E)$ , matriz de distâncias  $W$ , onde  $w(i, j)$ , para cada aresta  $(i, j) \in E$ ,  $1 \leq i, j \leq n$ .

```

 $V' := \{v_1\}$ ;  $c(1) := 0$ 
para  $v_i \in V - V'$  efetuar
     $c(i) := w(1, i)$ 
enquanto  $V' \neq V$  efetuar
    escolher  $v_j \in V - V'$  que maximiza o valor  $c(j)$ 
     $V' := V' \cup \{v_j\}$ 
    para  $v_i \in V - V'$  efetuar
         $C(i) := \max\{c(i), c(j) + w(i, j)\}$ 

```

*Pergunta:* Esse algoritmo determina o comprimento do caminho máximo simples de  $v_1$  para cada vértice  $v_i \in V$ ? Justifique a resposta.

- 7.5 Através de um exemplo, mostrar que o algoritmo de Dijkstra não obtém necessariamente a resposta correta, caso o digrafo contenha distâncias negativas.
- 7.6 Modificar o algoritmo de Dijkstra, de modo a encontrar os comprimentos dos caminhos mínimos de todos os vértices para um vértice fixo.
- 7.7 Modificar o algoritmo de Dijkstra, de forma a encontrar os caminhos mínimos que sejam os menores lexicograficamente do vértice fixo para os demais, dentre todos os caminhos mínimos existentes.
- 7.8 Seja  $D(V, E)$  um digrafo sem ciclos negativos,  $V = \{v_1, \dots, v_n\}$  e  $\ell_k$  um inteiro,  $0 \leq \ell_k \leq n - 1$ . Escrever um algoritmo para determinar o comprimento do caminho mínimo de  $v_1$  a  $v_k$ , contendo *exatamente*  $\ell_k$  arestas.
- 7.9 Escrever uma implementação do algoritmo de Bellman-Ford para determinar os comprimentos dos caminhos mínimos de um vértice  $v_1$  para todos os demais, em um grafo  $D(V, E)$ , sem ciclos negativos. Esta implementação deve terminar em tempo  $O(nm)$ ,  $n = |V|$  e  $m = |E|$ .
- 7.10 O que ocorre com a computação de caminhos mínimos entre cada par de vértices, através do Algoritmo 7.3, de Floyd, quando o grafo de entrada  $D$  contém ciclos negativos? Iluste através da descrição de um exemplo.
- 7.11 Modificar o algoritmo de Floyd, de modo a obter também todos os caminhos mínimos entre pares de vértices, além de seus comprimentos.
- 7.12 Seja o Algoritmo 7.4, para determinação dos  $k$ -ésimos menores caminhos de  $v_1$  para cada vértice  $v_i \in V$ , com possíveis vértices repetidos, em um grafo  $D$  sem ciclos negativos. Modificar o algoritmo de tal modo que cada vértice somente possa ocorrer duas vezes, no máximo, em cada caminho, e com a restrição de que, entre as duas possíveis ocorrências de um vértice  $v_i$  em um caminho, existam no máximo  $r$  outros vértices, para um dado valor inteiro  $r > 0$ .
- 7.13 O Algoritmo 7.5, para computar os  $k$ -ésimos caminhos simples de  $v_1$  para cada vértice  $v_j \in V$ , em um grafo  $D(V, E)$  sem ciclos negativos, estaria correto se  $D$  contivesse ciclos negativos? Em caso positivo, prove. Em caso negativo, apresentar um contra-exemplo.
- 7.14 Seja  $D$  um digrafo,  $D^+$  a sua expansão (Seção 7.8),  $v_k$  um vértice de  $D$ ,  $c(\ell, k)$  o comprimento do caminho mínimo do vértice origem  $v_1$  de  $D^+$  até  $v_k$ , contendo  $\ell$  arestas, no máximo, e  $n$  o número de vértices de  $D^+$ . Se  $c(n - 1, k) = c(n, k)$  então existe necessariamente um ciclo negativo no caminho de  $v_1$  a  $v_k$ ? Justificar a resposta dada.
- 7.15 Segundo a notação do exercício anterior, suponha que o vértice  $v_k$  satisfaça  $c(n - 1, k) \neq c(n, k)$ . Formular um algoritmo para, efetivamente, determinar um ciclo negativo contido no caminho de  $v_1$  a  $v_k$ .

- 7.16 Seja  $D(V, E)$  um digrafo,  $W$  sua matriz de distâncias, todas não negativas, e  $v_i, v_j \in V$  vértices arbitrários de  $D$ , onde  $(v_i, v_j) \notin E$ . Determinar o ciclo de comprimento mínimo em  $D$ , contendo  $v_i$  e  $v_j$
- 7.17 Seja  $D(V, E)$  um digrafo,  $W$  sua matriz de distâncias, com todos os valores não negativos, e  $v_i, v_j \in V$  vértices arbitrários distintos de  $D$ , onde  $(v_i, v_j) \notin E$ . Determinar dois caminhos de  $v_i$  para  $v_j$ , que não contenham vértices comuns além dos extremos, e cuja soma de seus comprimentos seja mínima.
- 7.18 Seja um conjunto  $P$  de países  $p_1, \dots, p_n$ , cada  $p_i$  com a sua moeda local  $m_i$ . Seja  $M$  a matriz  $n \times n$ , tal que sua entrada  $i, j$  corresponda à cotação do valor da moeda  $m_i$ , no país  $p_j$ , todos os câmbios considerados num mesmo instante,  $1 \leq i, j \leq n, i \neq j$ . Pede-se:

- (a) Verificar se através de câmbios sucessivos de moedas é possível aumentar um valor inicial investido no câmbio de moedas. Em caso positivo, determinar também:
- (b) A menor sequência de câmbios que produza lucro.
- (c) A sequência de câmbios que produza lucro máximo.

*Sugestão:* Modelar o problema como de detecção de ciclos negativos numa matriz de distâncias.

#### 7.19 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.

#### 7.20 POSCOMP 2012

Concernente aos algoritmos em grafos, relate a coluna da esquerda com a da direita.

Considerando as afirmativas das colunas esquerda e direita anteriores, assinale a alternativa que contém a associação correta.

- |                                 |  |
|---------------------------------|--|
| a) I-A, II-B, III-C, IV-D, V-E. |  |
| b) I-C, II-D, III-E, IV-A, V-B. |  |
| c) I-C, II-E, III-B, IV-A, V-D. |  |
| d) I-D, II-B, III-A, IV-C, V-E. |  |
| e) I-D, II-E, III-A, IV-B, V-C. |  |

I Ordenação Topológica (Topsort).	A	Toma como entrada um grafo orientado, utiliza basicamente a busca em profundidade e o conceito de grafo transposto para resolver o problema.
II Árvore Geradora Minimal (Prim).	B	Toma como entrada um grafo não orientado com pesos nas arestas, ordena as arestas por peso e escolhe as arestas de forma a não fechar ciclos para resolver o problema.
III Caminhos mais curtos (Dijkstra).	C	Toma como entrada um grafo orientado acíclico, utiliza basicamente busca em profundidade e rotulação de vértices para resolver o problema
IV Componentes fortemente conexas (CFC).	D	Toma como entrada um grafo não orientado com pesos nas arestas, utiliza basicamente busca em largura escolhendo arestas de menor peso para resolver o problema.
V Árvore Geradora Minimal (Kruskal).	E	Toma como entrada um grafo não orientado com pesos nas arestas, utiliza basicamente busca em largura escolhendo distâncias acumuladas de menor peso para resolver o problema.

### 7.21 UVA Online Judge 12144

Escreva um algoritmo para o seguinte problema:

Como hoje em dia muitos motoristas consultam aplicativos para determinar menores caminhos entre dois pontos, muitos trechos de ruas que fazem parte de menores caminhos estão se tornando mais congestionados, então deixa de ser interessante seguir esses caminhos. Seu patrão tem um compromisso urgente e pediu para você encontrar um caminho “quase mínimo”. Ele definiu esse caminho como o caminho de menor distância que não passa por nenhum trecho que faz parte de menores caminhos para o destino que ele vai. Você conhece o ponto inicial, o ponto final e todos os trechos de rua entre esses pontos, bem como o comprimento de cada trecho.

### 7.22 UVA Online Judge 10801

Escreva um algoritmo para o seguinte problema:

Um edifício tem  $n$  andares, numerados de 0 a  $n-1$ ,  $n < 100$ . Há  $p$  elevadores, que trafegam com velocidades diferentes e param em andares específicos, mas todos param no térreo(andar 0). Você não pode pegar as escadas e leva 1 minuto para trocar de elevador, caso precise fazer isso para alcançar o andar desejado. Dados  $n, p$ , os tempos de cada um dos elevadores para trafegar entre dois andares e os pontos de parada de cada elevador, determinar se é possível chegar ao andar  $n$  e, em caso positivo, qual o tempo mínimo para isso.

### 7.23 UVA Online Judge 10557

Escreva um algoritmo para o seguinte problema:

Um jogo de computador consiste em  $n$  quartos interligados por  $m$  portas unidirecionais. Um dos quartos é o de início e outro é o final. Cada quarto tem um nível de energia informado, que varia entre  $-100$  e  $+100$ . Em cada quarto que o jogador passar ele acumula o nível de energia desse quarto ao que já tinha. No quarto

inicial ele recebe uma energia de +100. Se ele entrar em um quarto e ficar com a energia negativa, ele morre. Dada a configuração dos quartos, quer-se saber se é possível o jogador alcançar o quarto de destino ou não.

#### 7.24 UVA Online Judge 1056

Escreva um algoritmo para o seguinte problema:

Dada a crescente interconexão mundial pelas redes sociais, especula-se que qualquer pessoa na Terra seja separada de qualquer outra por não mais que 6 graus de separação. O grau de separação é o número mínimo de relacionamentos que têm que ser considerados para conectar duas pessoas. Para um certo conjunto de pessoas, o grau máximo de separação é o maior grau de separação entre duas pessoas quaisquer do conjunto. Dadas  $n$  pessoas e seus  $m$  relacionamentos, indicar se o conjunto é conexo (isto é, se há uma cadeia de relacionamento entre qualquer par de pessoas do mesmo) e, caso seja, qual o grau máximo de separação do mesmo.

#### 7.25 UVA Online Judge 10926

Escreva um algoritmo para o seguinte problema:

Dado um conjunto de tarefas e as dependências entre elas, deseja-se saber qual a tarefa tem o maior número de dependências. Uma tarefa  $A$  depende de uma tarefa  $B$  se ela depender da tarefa  $B$  direta ou transitivamente. Assuma que não há dependências cíclicas.

### 7.11 Notas Bibliográficas

As equações de Bellman foram publicadas em Bellman(1958). O Algoritmo 7.1, para encontrar caminhos mínimos em digrafos com distâncias não negativas, devido a Dijkstra, foi publicado em Dijkstra(1959). Este mesmo artigo contém um algoritmo para construção da árvore geradora mínima de um grafo. Para o Algoritmo 7.2 (Bellman-Ford) as referências são Bellman(1958) e Ford(1956). O Algoritmo 7.3, de Floyd, foi apresentado em Floyd(1962), utilizando resultados descritos em Warshall(1962). Um outro algoritmo para determinar o caminho mais curto entre cada par de vértices, em um grafo em que todas as distâncias são positivas foi descrito por Spira(1973). Para este último, Spira mostrou que a complexidade média é de  $O(n^2 \log^2 n)$ . O Algoritmo 7.4, para a determinação de  $k$ -ésimos caminhos mínimos com possíveis repretações de vértices, é devido a Lawler(1976). Este algoritmo foi parcialmente baseado em resultados de Dreyfus(1969). Por outro lado, o Algoritmo 7.5 para  $k$ -ésimos caminhos mínimos simples foi originalmente descrito em Lawler(1972). Este último representa uma melhoria de um algoritmo anterior, devido a Yen(1971). O algoritmo para a determinação de ciclos negativos é uma extensão do algoritmo de Bellman-Ford, conforme mencionado. De um modo geral, os livros de Lawler(1976) e Kleinberg and Tardos(2006) são referências importantes para caminhos mínimos. Deve ser mencionado que existem outros algoritmos de maior eficiência, para determinados problemas de caminhos mínimos. Entre esses, mencionamos o algoritmo de Eppstein(1998) que encontra os  $k$ -ésimos caminhos mínimos, com possíveis vértices repetidos, entre dois dados vértices na complexidade  $O(m + n \log n + k)$ . Henzinger, Klein, Rao and Subramanian(1997) descreveram um algoritmo linear para determinar os caminhos mínimos de um vértice para os demais, em um grafo planar cujas distâncias são não negativas. Thorup(1999) formulou um algoritmo para encontrar os caminhos mínimos em grafos não direcionados com distâncias inteiras e positivas, em tempo linear.

# EMPARELHAMENTOS MÁXIMOS EM GRAFOS

## 8.1 Introdução

Seja  $G(V, E)$  um grafo não direcionado,  $|V| = n$  e  $|E| = m$ . Um *emparelhamento*  $M$  é um subconjunto de arestas duas a duas não adjacentes. As arestas pertencentes a  $M$  são ditas  *$M$ -emparelhadas*, ou simplesmente *emparelhadas*, enquanto que um vértice incidente a alguma aresta  $M$ -emparelhada é dito  *$M$ -saturado*, e aqueles não incidentes à arestas de  $M$  são  *$M$ -expostos*. Os grafos da Figura 8.1 e da Figura 8.2 ilustram emparelhamentos cujas arestas estão representadas em negrito. A aresta  $(1, 3)$  não está  $M$ -emparelhada na Figura 8.1(a), porém está na Figura 8.1(b). O vértice 6 está  $M$ -exposto na Figura 8.1(a), porém está  $M$ -saturado na Figura 8.1(b) e na Figura 8.2. Neste último, todos os vértices estão  $M$ -saturados.

Observe que um conjunto vazio define um emparelhamento. Por outro lado, se  $M \subseteq E$  é um emparelhamento qualquer  $M' \subseteq M$  também é.

O emparelhamento  $M$  é maximal se qualquer aresta não emparelhada por  $M$  possuir uma das extremidades em  $M$ . Isto é, não existe emparelhamento  $M'$ , tal que  $M' \supset M$ . Além disso, se  $M$  for o emparelhamento de maior cardinalidade do grafo, então  $M$  é *máximo*. Isto é, não existe emparelhamento  $M'$ , tal que  $|M'| > |M|$ . Finalmente, diz-se que  $M$  é *perfeito* quando todo vértice do grafo for saturado por  $M$ . Naturalmente, todo emparelhamento perfeito é máximo, e todo emparelhamento máximo é maximal. O problema do emparelhamento consiste em determinar o emparelhamento de cardinalidade máxima do grafo. A Figura 8.1(a) apresenta um emparelhamento maximal, porém não máximo do grafo, este último aparece na Figura 8.1(b). O emparelhamento do grafo da Figura 8.2 é perfeito, enquanto que o grafo da Figura 8.1 não admite emparelhamento perfeito.

Uma variação importante do problema do emparelhamento consiste em considerar grafos ponderados, ou seja, com pesos não negativos, associados as suas arestas. Nesse caso, definimos o *peso do emparelhamento*  $M$  como a soma dos pesos das arestas que compõem  $M$ . O objetivo consiste em determinar o emparelhamento de peso máximo. Observe-se que um emparelhamento de peso máximo não necessariamente possui cardinalidade máxima,

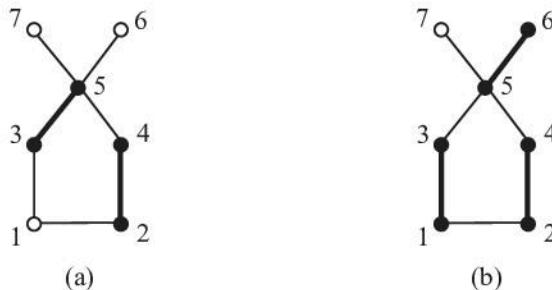


Figura 8.1: Emparelhamentos maximal e máximo

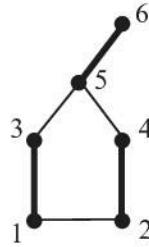


Figura 8.2: Emparelhamento perfeito

e reciprocamente. Por exemplo, o grafo ponderado  $P_4$  que consiste nos vértices  $\{v_1, v_2, v_3, v_4\}$  e arestas  $\{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}$ , com pesos 1, 3, 1, respectivamente, possui a aresta  $\{(v_2, v_3)\}$ , de peso 3, com peso máximo ponderado, o qual não é de cardinalidade máxima. Este último no conjunto de arestas  $\{(v_1, v_2), (v_3, v_4)\}$ .

De um modo geral, os problemas de emparelhamento tornam-se mais simples quando os grafos são bipartidos. Ao longo do presente capítulo, serão discutidos e/ou apresentados algoritmos para resolver os seguintes problemas de emparelhamento.

- (i) Determinar o emparelhamento máximo em grafos bipartidos.
- (ii) Determinar o emparelhamento máximo ponderado em grafos bipartidos.
- (iii) Determinar o emparelhamento máximo em grafos gerais.

## 8.2 Emparelhamentos Perfeitos

Nesta seção considera-se emparelhamentos perfeitos em grafos bipartidos, e o objetivo consiste em determinar condições necessárias e suficientes para a existência de tais emparelhamentos. Estas condições são estabelecidas pelo Teorema 8.1, devido a Hall.

### Teorema 8.1

Seja  $G(V, E)$  um grafo bipartido, com bipartição  $V_1 \cup V_2 = V$ ,  $|V_1| = |V_2|$ . Então  $G$  possui emparelhamento que satura todos os vértices de  $V_1$ , se e somente se  $|A(V'_1)| \geq |V'_1|$ , para todo  $V'_1 \subseteq V_1$ .

**Prova** Por hipótese, vale  $|A(V'_1)| \geq |V'_1|$ , para todo  $V'_1 \subseteq V_1$ . A prova é por indução na cardinalidade de  $V_1$ . Se  $|V_1| = 1$ , o resultado é trivial. Assuma o teorema verdadeiro se  $|V_1| \leq k$ , e considere agora um grafo  $G$  em que  $|V_1| = k + 1$ . Há duas alternativas:  $|A(V'_1)| > |V'_1|$ , para todo subconjunto próprio  $V'_1 \subset V_1$ , ou existe um subconjunto próprio  $V'_1 \subset V_1$  satisfazendo  $|A(V'_1)| = |V'_1|$ .

- (i) Seja  $|A(V'_1)| > |V'_1|$ , para todo  $V'_1 \subset V_1$ , escolha  $v_1 \in V'_1$ . Então  $|A(v'_1)| \geq 1$ , por hipótese. Escolha  $v_2 \in A(v'_1)$ , e seja  $H$  o grafo obtido pela remoção dos vértices  $v_1, v_2$  e das arestas a eles incidentes. Então  $H$  satisfaz à hipótese do teorema, pois como a alternativa (i) implica  $|A(V'_1)| > |V'_1|$ , a remoção de um vértice  $v_2$  de  $V_1$ , faz decrescer  $|A(V'_1)|$  no máximo de uma unidade, o que ainda assegura  $|A(V'_1)| \geq |V'_1|$ , para todo  $V'_1 \subseteq V$ . Pela hipótese de indução,  $H$  possui um emparelhamento que satura todo vértice de  $V_1 \setminus \{v_1\}$ . Adicionando-se a aresta  $(v_1, v_2)$  a este emparelhamento, obtem-se um emparelhamento de  $G$ , que satura todos os vértices de  $V_1$ .
- (ii) Seja  $V'_1 \subset V_1$ , tal que  $|A(V'_1)| = |V'_1|$ . Inicialmente, construir dois grafos bipartidos auxiliares  $H_1$  e  $H_2$ , com bipartição  $V'_1 \cup A(V'_1)$  e  $(V_1 \setminus V'_1) \cup (V_2 \setminus A(V'_1))$ , respectivamente. As arestas de  $H_1$  e  $H_2$  são exatamente as existentes em  $G$ , para pares de vértices correspondentes. Inicialmente, mostramos que ambos  $H_1$  e  $H_2$  satisfazem à hipótese do teorema. Para verificar  $H_1$ , considere qualquer subconjunto  $W \subseteq V'_1$ . Então as vizinhanças de  $W$ , em  $H_1$  e  $G$  coincidem, isto é,  $A_{H_1}(W) = A_G(W)$ . Como  $|A_G(W)| \geq |W|$ , segue-se que  $|A_{H_1}(W)| \geq |W|$ . Para verificar  $H_2$ , suponha, por absurdo, que existe um subconjunto  $Z \subseteq V_1 \setminus V'_1$  cuja vizinhança em  $H_2$  satisfaça  $|A_{H_2}(Z)| < |Z|$ . Isto implica que o subconjunto  $Z \cup V'_1$  possua vizinhança em  $G$  satisfazendo  $|A_G(Z \cup V'_1)| < |Z \cup V'_1|$ , pois  $A_G(Z \cup V'_1) = A_{H_2}(Z) \cup A_G(V'_1)$  e  $|A_G(V'_1)| = |V'_1|$ .

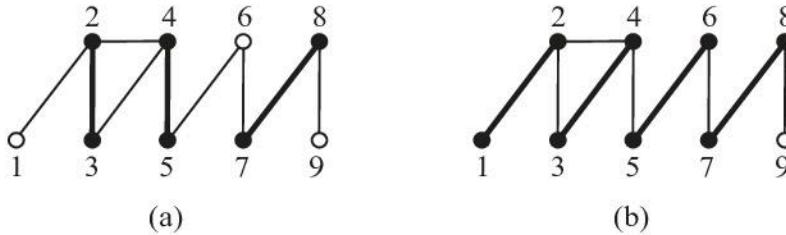


Figura 8.3: Caminhos alternante e aumentante

Logo,  $|A_{H_2}(Z)| \geq |Z|$ , e ambos  $H_1$  e  $H_2$  satisfazem à hipótese do teorema. Nesse caso, observando que  $V'_1$  é um subconjunto próprio de  $V_1$ , conclui-se que  $|V'_1|, |V_1 \setminus V'_1| \leq k$ . Este fato permite aplicar diretamente a hipótese de indução para ambos  $H_1$  e  $H_2$ , e concluir que  $H_1$  e  $H_2$  admitem emparelhamentos que saturam  $V'_1$  e  $V_1 \setminus V'_1$ , respectivamente. Finalmente, um emparelhamento de  $G$  que sature todos os vértices de  $V_1$ , pode ser obtido pela união dos emparelhamentos obtidos de  $H_1$  e  $H_2$ .

O argumento apresentado prova a suficiência do teorema. A prova de necessidade é imediata. Se  $V_1$  contém um subconjunto  $V'_1$  satisfazendo  $|A(V'_1)| < |V'_1|$ , é claro que não há como escolher  $|V'_1|$  vértices distintos em  $A(V'_1)$ , onde cada um deles é adjacente somente a um vértice distinto de  $V'_1$ . Então vale o teorema. ■

### Corolário 8.1

(Hall) Seja  $G(V, E)$  um grafo bipartido, com partições  $V_1 \cup V_2 = V$ ,  $|V_1| = |V_2|$ . Então  $G$  possui emparelhamento perfeito se e somente se  $|A(V_1)| \geq |V_1|$ .

**Prova** Aplicando-se o Teorema 8.1 para  $G$ , obtém-se um emparelhamento  $M$  que satura todos os vértices de  $V_1$ , ou seja, de cardinalidade  $|V_1|$ . Como  $|V_1| = |V_2|$ ,  $M$  também satura todos os vértices de  $V_2$ , ou seja,  $M$  é perfeito.

## 8.3 Caminhos Alternantes

Nesta seção, será apresentado o conceito de caminhos alternantes, o qual é central no estudo de algoritmos para a determinação de emparelhamentos máximos.

Seja  $G$  um grafo geral e  $M$  um emparelhamento de  $G$ . Um *caminho  $M$ -alternante* em  $G$  é um caminho cujas arestas estão alternadamente em  $E - M$  e  $M$ . Um caminho  $M$ -alternante cujos vértices extremos são ambos  $M$ -expostos é dito  *$M$ -aumentante*. Observe que um caminho  $M$ -aumentante possui uma quantidade par de vértices. A Figura 8.3(a) ilustra os conceitos apresentados. O emparelhamento considerado é  $M = \{(2, 3), (4, 5), (7, 8)\}$ . O caminho 1, 2, 3, 4, 5 é  $M$ -alternante, porém não  $M$ -aumentante, enquanto 1, 2, 3, 4, 5, 6 é um caminho  $M$ -aumentante. Através da operação de diferença simétrica entre as arestas de  $M$  e as do caminho  $M$ -aumentante apresentadas, transforma-se o emparelhamento  $M$ , no emparelhamento  $M' = \{(1, 2), (3, 4), (5, 6), (7, 8)\}$ , de cardinalidade uma unidade maior. Os emparelhamentos  $M$  e  $M'$  são ilustrados na Figura 8.3(a) e na Figura 8.3(b), respectivamente.

Observe que determinar emparelhamentos que sejam maximais é uma questão bastante simples; pode ser resolvida através de algoritmo guloso, em tempo linear no tamanho do grafo. A determinação de emparelhamentos máximos, contudo, é mais envolvente. Para esta última tarefa, aplica-se o Teorema 8.2 devido a Berge, que caracteriza emparelhamentos de cardinalidade máxima através de caminhos aumentantes.

### Teorema 8.2

(Berge) Seja  $G(V, E)$  um grafo qualquer e  $M$  um emparelhamento de  $G$ . Então  $M$  possui cardinalidade máxima se e somente se  $G$  não possui caminho  $M$ -aumentante.

**Prova** Seja  $M$  um emparelhamento em  $G$ . Suponha que  $G$  possui um caminho  $M$ -aumentante  $P$ . Observe que  $P$  tem, por definição, um número par de vértices, digamos  $P = v_0, v_1, \dots, v_{2k+1}$ .

Defina  $M' \subseteq E$  através de  $M' = M \setminus \{(v_1, v_2), (v_3, v_4), \dots, (v_{2k-1}, v_{2k})\} \cup \{(v_0, v_1), (v_2, v_3), \dots, (v_{2k}, v_{2k+1})\}$ .

Temos que  $M'$  é um emparelhamento em  $G$  com  $|M'| = |M| + 1$  arestas e, portanto,  $M$  não tem cardinalidade máxima em  $G$ . Logo, se  $M$  possui cardinalidade máxima em  $G$ , então  $G$  não possui caminho  $M$ -aumentante.

Por outro lado, suponha que  $M$  não tem cardinalidade máxima em  $G$ . Seja  $M'$  um emparelhamento de cardinalidade máxima. Logo  $|M'| > |M|$ . Denote por  $M \Delta M'$  a diferença simétrica de  $M$  e  $M'$ , i.e., o conjunto das arestas que estão em  $M \cup M'$  mas não estão em  $M \cap M'$ . Seja  $H = G[M \Delta M']$ . Cada vértice em  $H$  tem grau 1 ou 2, já que pode ser incidente a no máximo uma aresta de  $M$  e a uma aresta de  $M'$ . Logo cada componente conexa de  $H$  é um ciclo par com arestas alternadamente em  $M$  e  $M'$  ou um caminho com arestas alternadamente em  $M$  e  $M'$ . Mas  $H$  tem mais arestas de  $M'$  do que de  $M$  e, portanto, alguma componente que é um caminho  $Q$  em  $H$  deve começar e terminar com arestas de  $M'$ . Como a origem e o término de  $Q$  são  $M$ -saturados em  $H$ , segue que são vértices  $M$ -expostos em  $G$ . Logo  $Q$  é o caminho  $M$ -aumentante procurado. Portanto, se  $M$  não tem cardinalidade máxima em  $G$ , então  $G$  possui caminho aumentante em relação a  $M$ . ■

Um algoritmo natural para encontrar um emparelhamento de cardinalidade máxima decorre do Teorema 8.2. Iniciar com um emparelhamento vazio e, repetidamente, aumentar a cardinalidade do emparelhamento corrente através do uso sucessivo de caminhos aumentantes. Observe que a existência de um caminho aumentante  $P$  define através da operação diferença simétrica um novo emparelhamento  $M' = M \Delta E_P$ , onde  $E_P$  é o conjunto das arestas de  $P$ , e  $|M'| = |M| + 1$ . Este processo de obter sucessivos caminhos aumentantes termina com um emparelhamento de cardinalidade máxima porque:

- (i) A cardinalidade do emparelhamento de cardinalidade máxima é finita.
- (ii) Cada iteração aumenta de uma unidade a cardinalidade do emparelhamento corrente.

A complexidade do algoritmo anterior é função da procura por caminhos aumentantes. Observe que, no capítulo sobre algoritmos para fluxo máximo em redes, foi justamente o estudo da procura por caminhos aumentantes na rede residual que permitiu a descrição de algoritmos cada vez mais eficientes para aquele problema.

## 8.4 Grafos Bipartidos sem Pesos: Cardinalidade Máxima

Nesta seção, descreveremos um algoritmo para encontrar um emparelhamento máximo em um grafo bipartido.

O algoritmo para determinar um emparelhamento máximo, apresentado a seguir, é do tipo guloso, descrito na seção anterior. Iniciando-se por um emparelhamento qualquer  $M$ , iterativamente, buscar um caminho  $M$ -aumentante  $P$  de  $M$ , e definir  $M := M \Delta E_P$ , onde  $E_P$  representa o conjunto das arestas de  $P$ . Se  $G$  não mais admitir caminho aumentante, em relação a  $M$ , então o Teorema 8.2 assegura que  $M$  possui cardinalidade máxima.

Resta, agora a questão de como encontrar caminhos  $M$ -aumentantes em  $G$ . Para tal, utiliza-se um grafo direcionado auxiliar  $D(M)$ , obtido pela orientação das arestas de  $G$ , da seguinte maneira. Para um emparelhamento  $M$  do grafo  $G(V, E)$  com bipartição  $V = V_1 \cup V_2$ , direcionar cada aresta  $(v_1, v_2) \in E$ ,  $v_1 \in V_1$  e  $v_2 \in V_2$ , do seguinte modo

Se  $(v_1, v_2) \in M$ , então direcionar  $(v_1, v_2)$  de  $v_1$  para  $v_2$ ,

Caso contrário, direcionar de  $v_2$  para  $v_1$ .

O grafo assim obtido é o grafo direcionado  $D(M)$ . A sua utilização no algoritmo guloso de emparelhamento se baseia no seguinte lema, que indica como encontrar caminhos  $M$ -aumentantes de forma eficiente no grafo.

### Lema 8.1

Existe um caminho  $M$ -aumentante em  $G(V_1 \cup V_2, E)$  se e somente se existir um caminho (direcionado) em  $D(M)$ , entre dois vértices  $M$ -expostos,  $u, w \in V$ , iniciado por  $u \in V_2$  e terminado por  $w \in V_1$ .

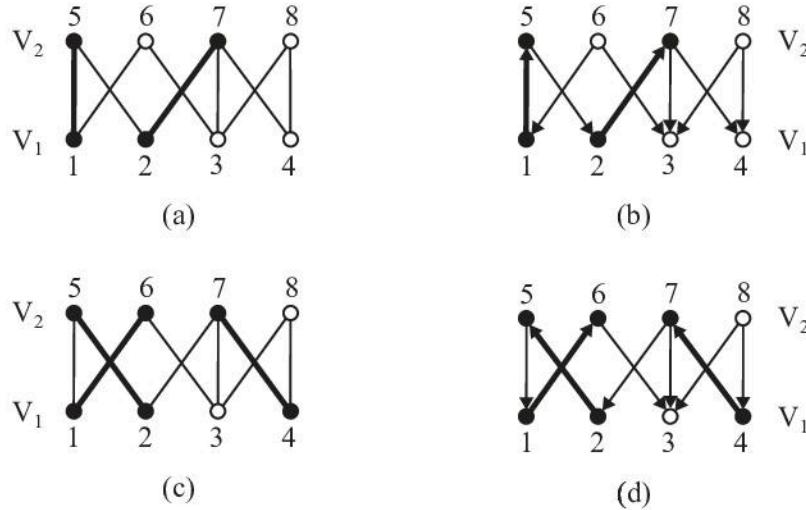


Figura 8.4: Aplicação do Lema 8.1

**Prova** Suponha que  $G$  admite um caminho  $M$ -aumentante  $P$  com vértices  $v_1, v_2, \dots, v_{k-1}, v_k$ . Então  $v_1$  e  $v_k$  são vértices  $M$ -expostos pelo emparelhamento  $M$  de  $G$ , tal que  $(v_1, v_2), (v_3, v_4), \dots, (v_{k-1}, v_k) \notin M$ , enquanto  $(v_2, v_3), (v_4, v_5), \dots, (v_{k-2}, v_{k-1}) \in M$ . Denote  $v_1 = u$  e  $v_k = w$ . Como  $G$  é bipartido,  $v_i$  e  $v_{i+1}$  pertencem a partes distintas da bipartição de  $G$ ,  $1 \leq i < k$ . Como  $v_1, \dots, v_k$  é um caminho aumentante,  $k$  é par. Consequentemente,  $u$  e  $w$  pertencem a partes distintas da bipartição. Além disso, todos os vértices  $M$ -expostos de  $V_1$  possuem grau de saída 0. Logo,  $u = v_1$  e  $w = v_k$ . A prova da recíproca é similar. ■

Como exemplo, considere o grafo bipartido  $G(V_1 \cup V_2, E)$  e o emparelhamento  $M$  ilustrado na Figura 8.4(a). O grafo direcionado  $D(M)$  correspondente aparece na Figura 8.4(b). O grafo  $D(M)$  contém o caminho direcionado  $P$  com vértices  $6, 1, 5, 2, 7, 4$  do vértice  $6 \in V_2$  para o vértice  $4 \in V_1$ , ambos  $M$ -expostos. Logo  $P$  é um caminho  $M$ -aumentante. Efetuando a operação de diferença simétrica, obtemos  $M := M \Delta E_P = \{(1, 5), (2, 7)\} \Delta \{(1, 6), (1, 5), (2, 5), (2, 7), (4, 7)\} = \{(1, 6), (2, 5), (4, 7)\}$ , o que resulta no aumento de uma unidade em  $M$ . A Figura 8.4(c) ilustra o novo emparelhamento, e a Figura 8.4(d) o novo digrafo  $D$ . Observe que este último contém ainda o caminho  $M$ -aumentante  $P$  do vértice  $8 \in V_2$  para  $3 \in V_1$ , ambos  $M$ -expostos. A diferença simétrica  $M := M \Delta E_P$  conduz então ao novo emparelhamento  $M = \{(1, 6), (2, 5), (3, 8), (4, 7)\}$ , o qual não contém caminhos do tipo procurado. Logo, o valor final de  $M$  é um emparelhamento máximo, na realidade perfeito.

O algoritmo para determinar emparelhamentos máximos está agora aparente. Iterativamente, aplicar o Lema 8.1, em cada passo construindo o grafo direcionado  $D(M)$ , obtendo caminhos aumentantes. O Algoritmo 8.1 descreve o processo.

---

**Algoritmo 8.1:** Emparelhamento cardinalidade máxima – Grafos bipartidos

**Dados:** Grafo bipartido  $G(V, E)$ ,  $V = V_1 \cup V_2$

$M := \emptyset$

Construir  $D(M)$

**enquanto** existir caminho  $P$  em  $D(M)$  de um vértice  $M$ -exposto  $u \in V_2$  para outro vértice  $M$ -exposto  $w \in V_1$   
**efetuar**

$M := M \Delta E_P$

Construir  $D(M)$

**retornar**  $M$

---

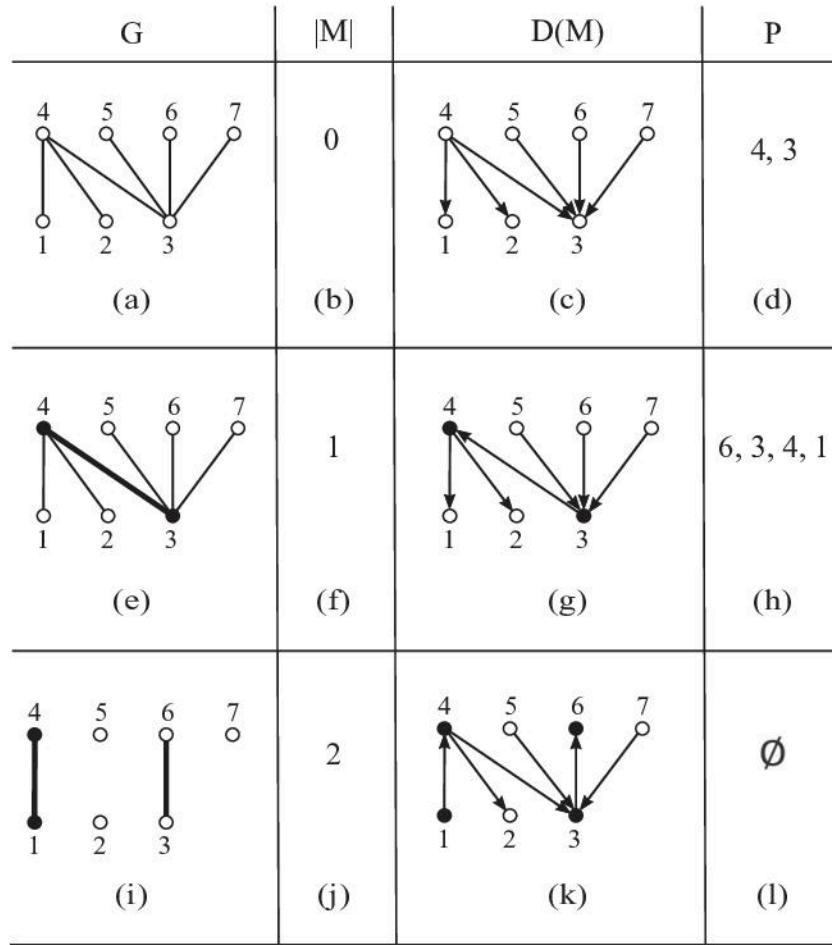


Figura 8.5: Exemplo de aplicação do Algoritmo 8.1

A procura dos caminhos  $M$ -aumentantes através do grafo direcionado  $D(M)$  pode ser realizada utilizando busca em profundidade iniciada por algum vértice  $M$ -exposto  $u \in V_2$ . Se a busca atingir algum vértice  $M$ -exposto  $w \in V_1$  um caminho  $M$ -aumentante terá sido encontrado.

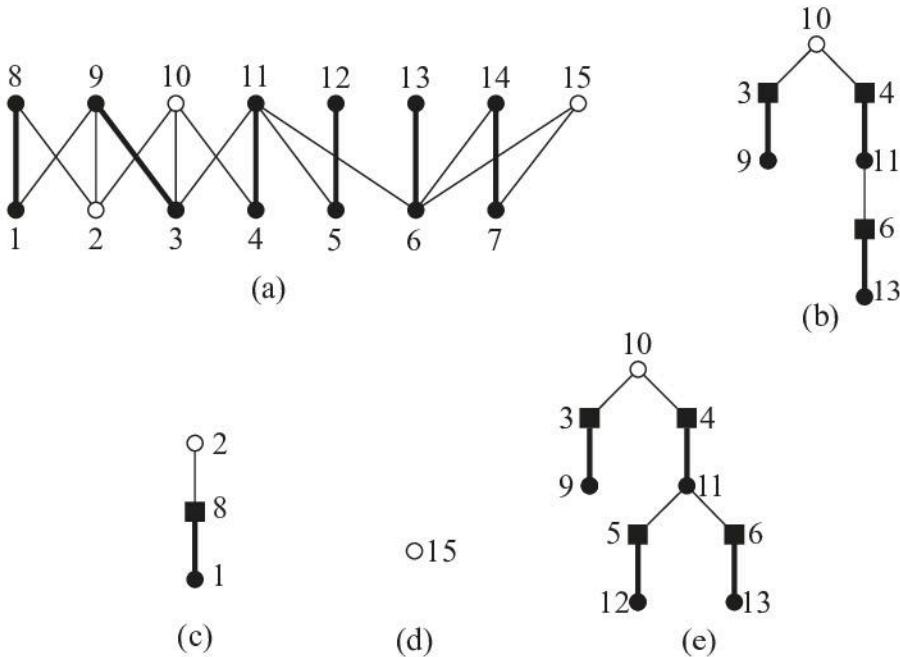
A determinação de complexidade é imediata. A construção de  $D(M)$ , a procura do caminho  $M$ -aumentante  $P$ , ambos requerem  $O(m)$  passos. A computação da diferença simétrica pode ser realizada em tempo  $O(|M| + |E_P|) = O(n)$ . Assim, cada iteração do algoritmo requer  $O(m)$  passos. Como a cardinalidade máxima de um emparelhamento é  $\lfloor \frac{n}{2} \rfloor$ , o número de iterações do algoritmo é  $O(n)$ . Logo, a complexidade final é  $O(nm)$ .

A Figura 8.5 apresenta um exemplo completo de aplicação do Algoritmo 8.1. O grafo de entrada aparece na Figura 8.5(a). O emparelhamento inicial  $M$  é igual a  $\emptyset$ , cuja cardinalidade nula é transcrita na Figura 8.5(b). O grafo direcionado  $D(M)$  correspondente é ilustrado na Figura 8.5(c), e o caminho  $P$  escolhido na Figura 8.5(d). O emparelhamento resultante da operação  $M := \emptyset \Delta (4, 3) = \{(4, 3)\}$  aparece na Figura 8.5(e), e assim por diante. O emparelhamento final máximo está na Figura 8.5(i), de cardinalidade 2, tendo em vista que o digrafo  $D(M)$  da Figura 8.5(k) não contém caminho entre dois vértices distintos  $M$ -expostos.

## 8.5 O Método Húngaro

Um método clássico para a determinação de emparelhamentos máximos em grafos bipartidos é denominado *Método Húngaro*, descrito nesta seção. Ele é utilizado para a determinação de caminhos aumentantes.

Seja  $G(V, E)$  um grafo e  $M$  um emparelhamento de  $G$ . Uma árvore enraizada  $T$ , formada por vértices e arestas de  $G$ , é denominada  $M$ -alternante, quando

Figura 8.6: Floresta  $M$ -alternante

- $T$  contém exatamente um vértice  $M$ -exposto, a raiz  $r_T$  de  $T$ .
- Para cada nó  $v$  de  $T$ , o caminho de  $r_T$  a  $v$  em  $T$  é  $M$ -alternante.
- Se  $v \neq r_T$  é uma folha e  $w$  é o pai de  $v$ , em  $T$ , então  $(w, v) \in M$ .

Como exemplo, a árvore da Figura 8.6(b) é uma árvore  $M$ -alternante do grafo e o emparelhamento  $M$  é ilustrado na Figura 8.6(a). A raiz 10 é o único vértice  $M$ -exposto de  $T$ .

Um nó  $v$  de uma árvore  $M$ -alternante  $T$  de  $G$  é denominada par (ímpar) se a sua distância à raiz for par (ímpar). No exemplo da Figura 8.6(b), os nós impares são representados por quadrados, e os pares por círculos. Uma floresta  $M$ -alternante  $F$  de um grafo  $G$  é a união disjunta de árvores  $M$ -alternantes, tal que cada vértice  $M$ -exposto é raiz de alguma árvore de  $F$ . Denote por  $\text{par}(F)$  e  $\text{ímpar}(F)$  os conjuntos dos nós pares e ímpares das árvores que compõem  $F$ , respectivamente. Como exemplo, a árvore da Figura 8.6(b), da Figura 8.6(c) e da Figura 8.6(d) constituem uma floresta  $M$ -alternante para o grafo e emparelhamento da Figura 8.6(a).

As florestas alternantes podem ser utilizadas para construir caminhos  $M$ -aumentantes, como a seguir.

Seja  $G(V, E)$  um grafo,  $M$  um emparelhamento de  $G$ , e  $F$  uma floresta  $M$ -alternante, formada por árvores  $T_1, \dots, T_k$ . Sejam  $u, v \in V$ , tais que  $(u, v) \in E$ ,  $u \in \text{par}(F)$  e  $v \notin \text{ímpar}(F)$ . As seguintes alternativas podem ocorrer.

**Caso 1:**  $v \notin \text{par}(F)$ .

Então  $v$  não está em  $F$ . Além disso,  $v$  é  $M$ -emparelhado, pois  $v$  não é raiz de alguma árvore de  $F$ . Mas  $(u, v) \notin M$ . Seja  $T \in F$  a árvore de  $F$  que contém  $u$ . Então podemos aumentar a árvore  $T$  adicionando as arestas  $(u, v)$  e  $(v, v')$  a  $T$ , onde  $v'$  é o vértice  $M$ -emparelhado a  $v$ . Esta operação é denominada **AUMENTAR( $F$ )**.

**Caso 2:**  $v \in \text{par}(F)$ .

Sejam  $T_u$  e  $T_v$  as árvores de  $F$  que contêm  $u$  e  $v$ , respectivamente.

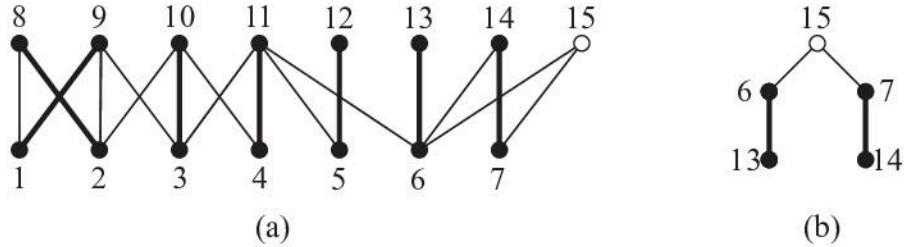


Figura 8.7: Floresta húngara

**Caso 2.1:**  $T_u \neq T_v$ 

Nesse caso, podemos obter um caminho  $M$ -aumentante em  $G$ , da seguinte maneira. Tomar o caminho de  $r_{T_u}$  até  $u$  em  $T_u$ ; acrescentar  $u$  a aresta  $(u, v)$ ; e finalmente, acrescentar  $v$ , bem como o caminho de  $v$  a  $r_{T_v}$  em  $T_v$ . Obtém-se desta forma um caminho  $M$ -aumentante em  $G$ , de  $r_{T_u}$  a  $r_{T_v}$ . Esta operação é denominada **AUMENTAR( $M$ )**.

**Caso 2.2:**  $T_u = T_v$ 

Então  $u$  e  $v$  são vértices pares,  $M$ -emparelhados, pertencentes a uma mesma árvore. Como consequência a aresta  $(u, v)$  não pertence à árvore. Este caso não pode ocorrer se  $G$  for bipartido, pois implica na existência de um ciclo ímpar.

Se não existem vértices  $u, v \in V$ , tais que  $(u, v) \in E$ ,  $u \in \text{par}(F)$  e  $v \notin \text{ímpar}(F)$ , então a floresta  $F$  é denominada *húngara*. Em particular, se  $M$  é um emparelhamento perfeito, não há vértices  $M$ -expostos. Nesse caso, a floresta  $M$ -alternante  $F$  correspondente é vazia, ou seja, é húngara.

Como exemplo, considere o grafo  $G$  e o emparelhamento  $M$  vistos na Figura 8.6(a), bem como a floresta  $M$ -alternante formada pelas árvores mostradas na Figura 8.6(b), na Figura 8.6(c) e na Figura 8.6(d). Considere as seguintes situações.

Se  $u = 11$  e  $v = 5$ , então  $(u, v) \in E$ ,  $u \in \text{par}(F)$  e  $v \notin \text{ímpar}(F)$ , que corresponde ao Caso 1. Como  $v \notin \text{par}(F)$ , a árvore apresentada na Figura 8.6(b), que contém o vértice 11 cresce com a adição das arestas  $(11, 5)$  e  $(5, 12)$ . Ou seja, a árvore da Figura 8.6(b) é substituída pela árvore da Figura 8.6(e). Ocorre então a operação **AUMENTAR( $F$ )**.

Se  $u = 9$  e  $v = 1$ , então  $(u, v) \in E$ ,  $u, v \in \text{par}(F)$ , o que corresponde ao Caso 2. As árvores da Figura 8.6(e) e da Figura 8.6(c) correspondem a  $T_u$  e  $T_v$ , respectivamente, o que significa  $T_u \neq T_v$ , isto é, Caso 2.1. O caminho de 10 a 9, na Figura 8.6(e), unido à aresta  $(9, 1)$ , unido ao caminho de 1 a 2 na Figura 8.6(c), produz o caminho  $P$  formado pelos vértices 10, 3, 9, 1, 8, 2 que é  $M$ -aumentante. Com isso, o emparelhamento  $M$  é substituído pelo emparelhamento  $M' = M \Delta P$ , de cardinalidade uma unidade maior. Ocorre então a operação **AUMENTAR( $M$ )**. A Figura 8.7(a) ilustra o grafo  $G$  e o novo emparelhamento  $M'$ , enquanto que na Figura 8.7(b) aparece uma floresta  $M'$ -alternante  $F$  correspondente a  $G$  e a  $M'$ . Em particular, os vértices pares de  $F$  são 15, 13 e 14. Todos vizinhos de 15, 13 e 14 são vértices pertencentes a  $\text{ímpar}(F)$ . Ou seja, não existem  $u, v \in V$ ,  $(u, v) \in E$ ,  $u \in \text{par}(F)$ , e  $v \notin \text{ímpar}(F)$ . Consequentemente, a floresta  $M$ -alternante da Figura 8.7(b) é húngara.

O corolário seguinte é uma consequência do Teorema 8.2, que relaciona emparelhamentos máximos e caminhos aumentantes.

**Corolário 8.2**

Seja  $M$  um emparelhamento de  $G(V, E)$  e  $F$  uma floresta  $M$ -alternante húngara. Então  $M$  possui cardinalidade máxima.

**Prova** Para provar o Corolário 8.2, podemos utilizar um argumento por absurdo. Suponha que  $M$  não seja máximo. Então pelo Teorema 8.2, grafo  $G$  possui caminho  $M$ -aumentante  $v_1, v_2, \dots, v_{k-1}, v_k$ . Então  $v_1$  e  $v_k$  são raízes de árvores  $M$ -alternantes da floresta  $F$ . Nesse caso, existem vértices  $v_i, v_{i+1}$ , tais que  $v_1, \dots, v_i$  é um caminho  $M$ -alternante na árvore  $T_{v_1}$ , enquanto que  $v_k, v_{k-1}, \dots, v_{i+1}$  é caminho  $M$ -alternante em  $T_{v_k}$ . Logo, os vértices  $v_i, v_{i+1}$  satisfazem  $(v_i, v_{i+1}) \in E$ ,  $v_i \in \text{par}(F)$  e  $v_{i+1} \notin \text{ímpar}(F)$ . Isto é,  $F$  não é húngara, uma contradição. ■

Os Casos 1 e 2, descritos nesta seção, juntamente com o Corolário 8.2 conduzem a um algoritmo, denominado *algoritmo húngaro*, para a construção de uma floresta húngara de um grafo bipartido e, consequentemente, para a obtenção de um emparelhamento máximo.

Seja  $G(V, E)$  o grafo bipartido,  $V = \{v_1, \dots, v_n\}$ . O Algoritmo 8.2 descreve o processo. Partindo de um emparelhamento  $M$  de  $G$ , inicialmente vazio e de uma floresta  $M$ -alternante  $F$  correspondente, o algoritmo iterativamente considera arestas  $(u, v) \in E$ , tais que  $u \in \text{par}(F)$  e  $v \notin \text{ímpar}(F)$ . Se existem vértices  $u, v$  nessas condições, o algoritmo ou cresce  $F$  através da inclusão das novas arestas (operação AUMENTAR( $F$ )) ou encontra um caminho  $M$ -aumentante  $P$  (operação AUMENTAR( $M$ )). Neste último caso, são redefinidos  $F$ ,  $\text{par}(F)$  e  $\text{ímpar}(F)$ . Se não existirem vértices  $u, v$  nas condições anteriores, a floresta  $M$ -alternante  $F$  correspondente é húngara, o emparelhamento  $M$  é máximo, e o algoritmo termina.

---

**Algoritmo 8.2:** Emparelhamento cardinalidade máxima – Método Húngaro

---

**Dados:** grafo bipartido  $G(V, E)$

$M := \emptyset$ ; EXPOSTO :=  $V$

**para**  $v \in \text{EXPOSTO}$  **efetuar**

$T_v :=$  árvore contendo único vértice  $v$

$F := \{T_v | v \in \text{EXPOSTO}\}$

$\text{par}(F) := \text{EXPOSTO}$ ;  $\text{ímpar}(F) := \emptyset$

**enquanto** existir  $u, v \in V$ ,  $(u, v) \in E$ ,  $u \in \text{par}(F)$ ,  $v \notin \text{ímpar}(F)$  **efetuar**

**se**  $v \notin \text{par}(F)$  **então**

$v' :=$  vértice  $M$ -emparelhado a  $v$

$T :=$  árvore de  $F$  que contém  $u$

            adicionar as arestas  $(u, v), (v, v')$  a  $T$

$\text{par}(F) := \text{par}(F) \cup \{v'\}$

$\text{ímpar}(F) := \text{ímpar}(F) \cup \{v\}$

**caso contrário**

$T :=$  árvore de  $F$  que contém  $u$

$T' :=$  árvore de  $F$  que contém  $v$

$r :=$  raiz de  $T$ ;  $r' :=$  raiz de  $T'$

$P :=$  caminho em  $T$  de  $r$  a  $u$

$P' :=$  caminho em  $T'$  de  $v$  a  $r'$

$P'' :=$  concatenação de  $P$ , aresta  $(u, v)$ , e  $P'$

$M := M \Delta P''$

$\text{EXPOSTO} := \text{EXPOSTO} - \{r, r'\}$

**para**  $v \in \text{EXPOSTO}$  **efetuar**

$T_v :=$  árvore formada do único vértice  $v$

$F := \{T_v | v \in \text{EXPOSTO}\}$

$\text{par}(F) := \text{EXPOSTO}$ ;  $\text{ímpar}(F) := \emptyset$

**retornar** floresta húngara  $F$ , e o emparelhamento  $M$  máximo.

---

Para determinar a complexidade do algoritmo, observar que cada aplicação da operação AUMENTAR( $F$ ) pode ser realizada em tempo constante, mediante a utilização de uma estratégia eficiente para determinar a árvore de  $T$  que contém  $u$ . A operação AUMENTAR( $M$ ) pode ser realizada em  $O(n)$  passos, e no máximo  $O(n)$  vezes ao longo de todo o processo. Isto é, AUMENTAR( $M$ ) requer  $O(n^2)$  passos no total. Entre duas operações

AUMENTAR( $M$ ), cada aresta  $(u, v) \in E$  do bloco **enquanto** somente pode ser considerada uma vez, no máximo. Isto significa, que são realizadas  $O(nm)$  operações AUMENTAR( $F$ ) ao longo de todo o processo. Então a complexidade final é  $O(n^2 + nm)$ .

## 8.6 Emparelhamentos e Coberturas por Vértices

Há uma forte relação entre emparelhamentos máximos de grafos bipartidos e cobertura por vértices mínimas. Recordemos do Capítulo 2, que uma *cobertura por vértices* de um grafo  $G(V, E)$  é um subconjunto  $C \subseteq V$ , tal que cada aresta de  $G$  possui, pelo menos, uma extremidade em  $C$ . De um modo geral, o interesse é determinar a cobertura de cardinalidade mínima. Comparando-se as cardinalidades de uma cobertura  $C$  e de um emparelhamento  $M$ , tendo em vista que cada aresta de  $M$  é incidente a pelo menos um vértice distinto de  $C$ , concluímos que  $\min|C| \geq \max|M|$ .

Esta desigualdade é válida também para grafos gerais, não necessariamente bipartidos. Por exemplo, para cobrir as arestas de um triângulo são necessários dois vértices, enquanto que o emparelhamento máximo possui apenas uma aresta. Por outro lado, o emparelhamento ilustrado na Figura 8.1(b) consiste em 3 arestas, enquanto que este grafo admite uma cobertura  $C = \{1, 2, 5\}$ , de tamanho 3.

Um fato importante da teoria de emparelhamentos é que a desigualdade anterior, envolvendo emparelhamentos e coberturas, torna-se uma igualdade quando o grafo é bipartido. O teorema seguinte, devido a König, apresenta esta formulação.

### Teorema 8.3

(König) Seja  $G(V, E)$  um grafo bipartido com bipartição  $V = V_1 \cup V_2$ ,  $M$  um emparelhamento e  $C$  uma cobertura por vértices de  $G$ . Então  $\min|C| = \max|M|$ .

**Prova** Para mostrar a igualdade apresentada, reportamo-nos ao Algoritmo 8.1, da seção anterior. O algoritmo termina ao produzir o emparelhamento máximo  $M$  de  $G$ . Este algoritmo utiliza o digrafo  $D(M)$ , onde cada aresta  $(v_1, v_2) \in E$ , com  $v_1 \in V_1$  e  $v_2 \in V_2$ , é orientada de  $v_1$  para  $v_2$  se  $(v_1, v_2) \in M$ , e de  $v_2$  para  $v_1$  se  $(v_2, v_1) \notin M$ . Quando  $M$  é máximo, o digrafo  $D(M)$  não contém caminho de um vértice  $M$ -exposto de  $V_2$  para outro também  $M$ -exposto de  $V_1$ . Seja  $V' \subseteq V$  o conjunto de vértices alcançados por caminhos partindo de algum vértice  $M$ -exposto de  $V_2$ . Seja  $C \subseteq V$ , o conjunto de vértices definido como  $C = (V_2 - V') \cup (V_1 \cap V')$ .

Inicialmente, provamos que  $C$  é uma cobertura por vértices. Caso contrário, existe uma aresta  $(v_1, v_2) \in E$ ,  $v_1 \in V_1$ ,  $v_2 \in V_2$ , mas  $v_1, v_2 \notin C$ . Então  $v_1 \in V_1 \cap V'$  e  $v_2 \in V_2 - V'$ . Discutimos, em seguida, as alternativas possíveis para a aresta  $(v_1, v_2)$ . Suponha  $(v_1, v_2) \in M$ . Como  $v_2 \notin V'$ , a única maneira de alcançar  $v_2$  é através da aresta  $(v_1, v_2)$ , que está dirigida de  $v_1$  para  $v_2$ . Mas como  $v_1 \in V'$  isto implicaria em  $v_2 \in V'$ , contradição. Logo  $(v_1, v_2) \notin M$ . Isto significa que  $(v_1, v_2)$  está dirigida de  $v_2$  para  $v_1$ . Como  $v_1 \in V'$ , sabemos que  $v_1$  está  $M$ -emparelhado, caso contrário existiria um caminho  $M$ -aumentante iniciado por algum vértice  $M$ -exposto de  $V_2$  e terminado por  $v_1 \in V_1$ , o que contradiz  $M$  ser máximo. Por outro lado,  $v_2 \notin V'$  implica que  $v_2$  também está  $M$ -emparelhado. Então,  $v_1$  e  $v_2$  estão ambos  $M$ -emparelhados, o que significa  $(v_1, v_2) \in M$ , contradizendo  $(v_1, v_2) \notin M$ . Logo  $(v_1, v_2)$  não pode existir e, consequentemente,  $C$  é uma cobertura por vértices.

Resta mostrar a igualdade das cardinalidades  $|C| = |M|$ .

Sabemos que sempre vale  $|C| \geq |M|$ . Mas nesse caso, vamos provar que também vale  $|C| \leq |M|$ . Para tal, basta observar que  $C$  não pode conter vértices  $M$ -expostos, pois todos os vértices de  $(V_2 - V')$ , assim como os de  $(V_1 \cap V')$ , são saturados. Além disso, as arestas de  $M$  incidentes aos vértices de  $C$ , são distintas, o que implica  $|C| \leq |M|$ . Logo, vale a igualdade  $|C| = |M|$ . ■

Como exemplo, considere o grafo da Figura 8.5(a). O seu emparelhamento máximo  $M$ ,  $|M| = 2$  aparece na Figura 8.5(i), e o grafo  $D(M)$  correspondente na Figura 8.5(k). Neste último, verifica-se:  $V' = \{5, 3, 6, 7\}$ ;  $V_2 - V' = \{4\}$ ,  $V_1 \cap V' = \{3\}$ . Logo,  $C = \{4, 3\}$ . Então,  $|C| = 2$ , e  $C$  é uma cobertura por vértices mínima.

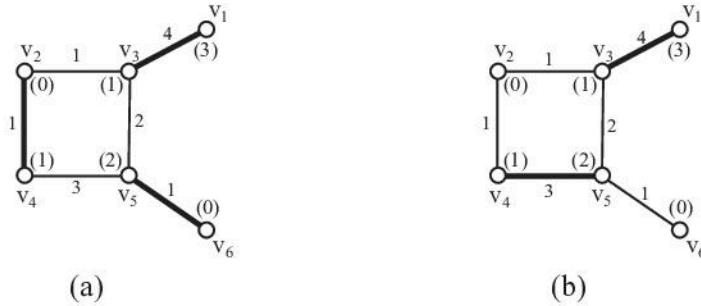


Figura 8.8: Emparelhamentos e cobertura ponderados

## 8.7 Grafos Bipartidos Ponderados

Nesta seção, consideramos grafos ponderados. O objetivo é descrever algoritmos para determinar emparelhamentos de peso máximo. Por outro lado, tratamos também do conceito de cobertura por vértices ponderada, onde o objetivo é produzir coberturas de peso mínimo. Os emparelhamentos ponderados máximos e as coberturas ponderadas mínimas conduzem a uma generalização do Teorema de König que será também apresentada.

Um *grafo ponderado*  $G(V, E)$  (nas arestas) é um grafo juntamente com uma função  $\omega$ , que associa um número real não negativo,  $\omega(e)$ , a cada aresta  $e \in E$ , denominado *peso da aresta*  $e$ . O *peso*  $\omega(M)$  de um emparelhamento  $M$  de  $G$ , é definido  $\omega(M) = \sum_{e \in M} \omega(e)$ . O problema ora considerado é determinar o emparelhamento  $M$  de  $G$  de peso (ponderado) máximo.

O algoritmo a ser apresentando utiliza também ponderação nos vértices, além das arestas. Similarmente, cada vértice  $v \in V$  estará associado a um número real não negativo, denominado *peso* de  $v$ . Utilizaremos uma ponderação especial de vértices, para definir uma extensão ponderada da ideia da cobertura por vértices. Seja  $G(V, E)$  um grafo com ponderação nas arestas  $\omega$ . Uma *cobertura por vértices ponderada* de  $G$  é uma função  $c$ , que atribui a cada vértice  $v_i \in V$  um real não negativo  $c(v_i)$  satisfazendo  $c(v_i) + c(v_j) \geq \omega(v_i, v_j)$  para cada aresta  $(v_i, v_j) \in E$ . O *peso* de uma cobertura por vértices ponderada  $c$  de  $G$  é definido como  $c(G) = \sum_{v_i \in V} c(v_i)$ . Dado um grafo com ponderação nas arestas, o objetivo é determinar uma cobertura por vértices ponderada de peso mínimo.

Seja  $M$  um emparelhamento ponderado do grafo  $G$ , e  $c$  uma cobertura por vértices ponderada de  $G$ . Então  $c(G) \geq \omega(M)$ . Para verificar esta desigualdade, basta somar as desigualdades  $c(v_i) + c(v_j) \geq \omega(v_i, v_j)$ , para todas as arestas  $(v_i, v_j) \in M$ , o que conduz a  $c(G) \geq \omega(M)$ .

Como exemplo, veja a Figura 8.8. O grafo ilustrado é ponderado nos vértices e arestas. Os pesos dos vértices  $c(v_i)$  e das arestas  $\omega(v_i, v_j)$  estão ambos indicados na figura, os que se encontram entre parênteses são pesos dos vértices. Observe que cada aresta  $(v_i, v_j)$  satisfaz  $c(v_i) + c(v_j) \geq \omega(v_i, v_j)$  o que significa que a ponderação dos vértices corresponde a uma cobertura por vértices ponderada. O peso da cobertura é  $c(G) = c(v_1) + \dots + c(v_6) = 3 + 0 + 1 + 1 + 2 + 0 = 7$ . O emparelhamento ilustrado na Figura 8.8(a) é  $M_1 = \{(v_1, v_3), (v_2, v_4), (v_5, v_6)\}$ , com peso  $4 + 1 + 1 = 6$ , enquanto o da Figura 8.8(b) é  $M_2 = \{(v_1, v_3), (v_4, v_5)\}$ , cujo peso é  $4 + 3 = 7$ . O emparelhamento  $M_1$  possui cardinalidade máxima, mas não peso máximo, enquanto que  $M_2$  é o emparelhamento de peso máximo. Observe que  $c(G) > \omega(M_1)$ , mas  $c(G) = \omega(M_2)$ .

Esse fato sempre ocorre para grafos bipartidos, assegurando a seguinte generalização do Teorema de König.

### Teorema 8.4

*Em um grafo bipartido ponderado, o peso do emparelhamento ponderado máximo é igual ao peso da cobertura por vértices ponderada mínima.*

Dado um grafo bipartido ponderado  $G(V, E)$ , descreveremos um algoritmo para determinar o emparelhamento ponderado máximo  $M$ , bem como uma cobertura por vértices ponderada mínima  $c$ . Iremos verificar que  $c(G) = \omega(M)$ , o que provará o teorema.

Sem perda de generalidade, podemos considerar que  $G(V, E)$ ,  $V = V_1 \cup V_2$ , é bipartido completo regular, pois se assim não for, podemos acrescentar vértices, para assegurar  $|V_1| = |V_2|$ , bem como arestas de peso nulo, as quais não influem nos pesos do emparelhamento máximo ou cobertura mínima. Nesse caso, consideramos  $|V_1| = |V_2|$ . Observe que com tais hipóteses, existe sempre um emparelhamento ponderado máximo, que satura todos os vértices de  $G$ , ou seja, um emparelhamento perfeito.

Em seguida, definimos a seguinte ponderação para os vértices de  $G$ .

$$c(v) = \begin{cases} \max\{\omega(v, v_2) | (v, v_2) \in E\}, & \text{se } v \in V_2 \\ 0, & \text{se } v \in V_1 \end{cases}$$

É claro que cada aresta  $(v_1, v_2) \in E$  satisfaz  $c(v_1) + c(v_2) \geq \omega(v_1, v_2)$ , o que significa que essa ponderação é uma cobertura por vértices ponderada, a qual é denominada *trivial*.

Para um grafo bipartido com ponderação nas arestas, e com cobertura ponderada  $c$ , para uma aresta  $(v_1, v_2) \in E$ , a diferença  $c(v_1) + c(v_2) - \omega(v_1, v_2)$  é denominada *excesso da aresta*  $(v_1, v_2)$ , e denotada por  $\varepsilon(v_1, v_2)$ .

Basicamente, a estratégia do algoritmo é a seguinte. Iniciando-se com uma cobertura trivial, o processo visa, iterativamente, diminuir os pesos de alguns vértices de  $V_2$ , e aumentar outros de  $V_1$ , até obter um emparelhamento perfeito  $M$ , e cujas arestas possuam excesso zero. Nessa condição estará garantida a igualdade  $c(G) = \omega(M)$ , conforme o Lema 8.2.

### Lema 8.2

Seja  $G(V, E)$  um grafo bipartido completo, regular, com ponderação nas arestas  $\omega$ , e com ponderação nos vértices  $c$ . Seja  $M$  um emparelhamento perfeito de  $G$ , tal que todas as arestas de  $M$  possuam excesso zero. Então  $c(G) = \omega(M)$ .

**Prova** Como  $\varepsilon(e) = 0$ , para cada aresta  $e = (v, w) \in M$ , sabemos que  $c(v) + c(w) = \omega(e)$ . Como  $M$  é um emparelhamento perfeito de  $G$ , a união dos vértices incidentes as arestas de  $M$  é o conjunto  $V$ , onde cada vértice aparece exatamente uma vez. Então

$$\omega(M) = \sum_{e \in M} \omega(e) = \sum_{e=(v,w) \in M} (c(v) + c(w)) = \sum_{v \in V} c(v) = c(G).$$

O algoritmo utiliza o seguinte conceito. O *subgrafo zero* do grafo  $G$ , é o subgrafo  $G_0(V_0, E_0)$ , induzido pelas arestas  $E_0$  de  $G$ , de excesso zero. O algoritmo constrói um emparelhamento  $M$  de cardinalidade máxima em  $G_0$ . Em seguida, efetua um procedimento similar ao da prova do Teorema de König. Seja  $D(M)$  o grafo direcionado correspondente a  $M$ , conforme utilizado no Algoritmo 8.1. Ou seja, para cada aresta  $(v_1, v_2) \in E$ ,  $v_1 \in V_1$  e  $v_2 \in V_2$  direcionar de  $v_1$  para  $v_2$ , se  $(v_1, v_2) \in M$  e de  $v_2$  para  $v_1$  quando  $(v_2, v_1) \notin M$ . Seja  $V' \subseteq V_0$  o subconjunto de vértices de  $G_0$ , alcançáveis em  $G_0$  através de caminhos de  $D(M)$ , iniciados por algum vértice  $M$ -exposto de  $V_0 \cap V_2$ . Seja  $E' \subseteq E$ , o conjunto de arestas de  $G$  com uma extremidade em  $V' \cap V_2$ , e outra em  $V_1 - V'$ . Seja  $V'_1 = V_1 \cap V'$  e  $V'_2 = V_2 \cap V'$ . Sabemos que  $E'$  não contém arestas de  $E_0$ , caso contrário  $M$  não seria de cardinalidade máxima, de acordo com o Teorema 8.2. Em seguida, determinar a aresta  $e \in E'$  de excesso mínimo  $\varepsilon$ . Resta agora atualizar os valores dos pesos dos vértices, como a seguir:

$$c(v) = \begin{cases} c(v) - \varepsilon, & \text{se } v \in V'_2 \\ c(v) + \varepsilon, & \text{se } v \in V'_1, \end{cases}$$

permanecendo inalterados os pesos dos demais vértices. Repetir o processo computando o novo grafo  $G_0$ , e assim iterativamente. O algoritmo termina quando o emparelhamento  $M$  encontrado for perfeito, o que garante  $\omega(M) = c(G)$ , segundo o Lema 8.2.

Formalmente, a correção do algoritmo pode ser descrita como se segue.

Ao final da iteração  $i$  do algoritmo, seja

$E_0(i)$  = conjunto das arestas de  $G$ , com excesso zero

$G_0(V_0, E_0)(i)$  = subgrafo de  $G$ , induzido pelas arestas  $E_0(i)$

$M_0(i)$  = emparelhamento de cardinalidade máxima de  $G_0(i)$

$c_0(i)$  = cobertura ponderada de vértices do grafo  $G_0(i)$ .

Introduzimos o Lema 8.3, para tratar do caso em que  $M_0(i)$  não é um emparelhamento perfeito. Neste caso, a iteração  $i$  não é a última, e o lema descreve as relações entre alguns parâmetros apresentados e os correspondentes à iteração  $i + 1$ .

### Lema 8.3

Se  $M_0(i)$  não é um emparelhamento perfeito, então, para  $i > 0$ ,

- (i)  $M_0(i) \subseteq E_0(i + 1)$
- (ii)  $c(G_0(i)) > c(G_0(i + 1))$
- (iii)  $V_2 \cap V_0(i) = V_2$
- (iv)  $V_1 \cap V_0(i) \subseteq V_1 \cap V_0(i + 1)$

**Prova** Considere a iteração  $i$ . Seja  $V' \subseteq V(G)$  o conjunto de vértices de  $V(G)$  alcançados por caminhos direcionados de  $D(M_0(i))$ , e iniciados por algum vértice de  $V'_2$  que seja  $M_0(i)$ -exposto de  $G_0(i)$ . Seja  $V'_1 = V_1 \cap V'$ ,  $V'_2 = V_2 \cap V'$  e  $(v_1, v_2) \in E_0(i)$ . Então  $v_1 \in V'_1$  se e somente se  $v_2 \in V'_2$ . Nesse caso, ou  $c(v_1)$ ,  $c(v_2)$  permanecem inalterados após a atualização dos pesos de  $G$ , ou então  $c(v_1)$  é incrementado de  $\varepsilon$ , e  $c(v_2)$  é decrementado de  $\varepsilon$ . Logo a diferença  $\varepsilon(v_1, v_2)$  se manterá. Isto é,  $(v_1, v_2) \in M_0(i)$  implica  $(v_1, v_2) \in E_0(i + 1)$ . Então (i) vale.

Para provar (ii), observe que a cada vértice  $v_1 \in V'_1$  corresponde um vértice  $v_2 \in V'_2$ , onde  $(v_1, v_2) \in M_0(i)$ . Contudo,  $V'_2$  contém um vértice  $M_0(i)$ -exposto sem correspondente em  $V'_1$ . Logo,  $|V'_1| < |V'_2|$ . Portanto há mais subtrações nos valores da ponderação de vértices do que somas. Logo, o peso total das ponderações dos vértices decresce a cada iteração, garantido  $c(G_0(i)) > c(G_0(i + 1))$ .

A afirmativa (iii) decorre do fato de que cada vértice de  $V_2$ , seja emparelhado ou exposto, está incluído em  $V_0(i)$ . Logo  $V_2 \cap V_0(i) = V_2$  e (iii) se verifica.

Finalmente, para mostrar (iv), observe que ao final da iteração  $i$ , cada vértice de  $V_1 \cap V_0(i)$  está emparelhado, ou é incidente a uma aresta de excesso 0, cuja outra extremidade é um vértice emparelhado. Nesse último caso, ele será incorporado a  $V_1 \cap V_0(i + 1)$ , o que prova a afirmativa (iv). ■

A cada iteração do algoritmo, aumenta a cardinalidade do emparelhamento, ou do conjunto  $|V_1 \cap V'|$ . Isto é, em cada iteração do bloco **repetir**, um vértice novo de  $V_1 \cap V_0$  entra no conjunto  $V'$ , pois uma nova aresta  $(v_1, v_2) \in E$ , onde  $v_1 \in V_1 - V'$  e  $v_2 \in V_2 \cap V'$  é adicionada a  $E_0$ , aquela aresta  $(v_1, v_2)$ , de excesso mínimo. Por outro lado, nenhum vértice de  $V_1 \cap V_0$  deixa de fazer parte de  $V'$ .

Como  $G$  é um grafo bipartido completo regular, concluímos que para uma certa iteração  $\ell$  do algoritmo  $V_0 = V_1 \cap V_0$ . Nesse caso, o emparelhamento obtido  $M_0(\ell)$  satura todos os vértices de  $G$ . Esta constitui a última iteração, e o algoritmo termina neste ponto, pois o emparelhamento  $M_0(\ell)$  é perfeito em  $G$ . Pelo Lema 8.3,  $c(G) = \omega(G)$ , o que completa a prova do Teorema 8.4.

O Algoritmo 8.3 descreve o processo, produzindo ao final um emparelhamento ponderado máximo  $M$  e uma cobertura por vértices ponderada mínima  $c$ .

Resta considerar ainda a determinação da complexidade do Algoritmo 8.3. A operação dominante, em termos de complexidade, do bloco

**repetir ... até que**

é a determinação do emparelhamento de cardinalidade máxima de  $G_0$ . Utilizando o Algoritmo 8.1, esta operação pode ser realizada em  $O(nm)$  passos. Contudo, de acordo com o Lema 8.3, as arestas do emparelhamento máximo  $M_0(i)$  obtido na iteração  $i$  são preservados na iteração  $i + 1$  como emparelhamento, embora não necessariamente máximo. Então, para determinar  $M_0(i + 1)$ , podemos partir de  $M_0(i)$ . Nesse caso, o número de passos realizados para a construção de todos os emparelhamentos máximos considerados é  $O(nm)$ . O lema seguinte determina o número máximo de iterações realizadas no bloco **repetir**.

**Algoritmo 8.3:** Emparelhamento ponderado máximo – Grafos bipartidos

**Dados:** grafo bipartido completo regular ponderado  $G(V, E)$ ,  $V = V_1 \cup V_2$ , peso real  $\omega(v_1, v_2) \geq 0$ , para cada aresta  $(v_1, v_2) \in E$ ,  $|V_1| = |V_2|$ .

**para**  $v \in V_2$  **efetuar**  
     $c(v) := \max\{\omega(v, v_2) | (v, v_2) \in E\}$

**para**  $v \in V_1$  **efetuar**  
     $c(v) := 0$

**para**  $(v_1, v_2) \in E$  **efetuar**  
     $\varepsilon(v_1, v_2) := c(v_1) + c(v_2) - \omega(v_1, v_2)$

$E_0 := \{(v_1, v_2) \in E | \varepsilon(v_1, v_2) = 0\}$

$G_0 :=$  subgrafo induzido em  $G$ , por  $E_0$

**repetir**  
     $M :=$  emparelhamento de cardinalidade máxima de  $G_0$   
    construir  $D(M)$   
    **se**  $M$  não for perfeito **então**  
         $V' := \{v \in V_0 | v$  é alcançável em  $D(M)$  de algum  $v_2 \in V_0 \cap V_2$ ,  $v_2$  exposto $\}$   
         $\varepsilon := \min\{\varepsilon(v_1, v_2) | v_1 \in V_1 - V'$  e  $v_2 \in V_2 \cap V'\}$   
        **para**  $v \in V_1 \cap V'$  **efetuar**  
             $c(v) := c(v) + \varepsilon$   
        **para**  $v \in V_2 \cap V'$  **efetuar**  
             $c(v) := c(v) - \varepsilon$   
        **para**  $(v_1, v_2) \in E$  **efetuar**  
             $\varepsilon(v_1, v_2) := c(v_1) + c(v_2) - \omega(v_1, v_2)$   
        **até que**  $M$  seja um emparelhamento perfeito em  $G$   
    **retornar** emparelhamento  $M$  e pesos  $c(v_i)$ ,  $v_i \in V$

---

**Lema 8.4**

O Algoritmo executa, no máximo,  $O(n^2)$  iterações do bloco **repetir**.

**Prova** Já sabemos que a cada iteração do algoritmo, aumenta a cardinalidade do emparelhamento, ou do conjunto  $|V_1 \cap V'|$ . Então no máximo após  $O(n)$  iterações, a cardinalidade do emparelhamento aumentará em uma unidade. Consequentemente, em  $O(n^2)$  iterações obtém-se um emparelhamento perfeito. ■

Como consequência deste lema, concluímos que o Algoritmo 8.3 pode ser implementado em  $O(n^2m)$ .

A Figura 8.9 ilustra um exemplo de aplicação do Algoritmo 8.3, para determinação do emparelhamento de peso máximo em um grafo bipartido ponderado completo. A Figura 8.9(a) descreve o grafo  $G$ , com pesos nas arestas, a matriz  $W$  de pesos. A cobertura trivial ponderada dos vértices  $C$ , aparece também nesta mesma figura. A Figura 8.9(b), a Figura 8.9(c) e a Figura 8.9(d) ilustram cada uma das três iterações efetuadas pelo algoritmo, até o término. Em cada uma delas, aparece o grafo  $G_0(i)$ , com o emparelhamento máximo  $M_0(i)$ , a matriz dos excessos das arestas  $\varepsilon(i)$ , e a cobertura ponderada  $c(i)$ , para  $i = 1, 2, 3$ . O emparelhamento obtido no passo 3, Figura 8.9(d), é perfeito, possui peso  $5 + 7 + 6 = 18$ , enquanto que a cobertura ponderada dos vértices possui peso  $5 + 5 + 3 + 3 + 2 + 0 = 18$

## 8.8 Grafos Gerais sem Peso

Nesta seção, descreveremos um algoritmo para encontrar o emparelhamento máximo em um grafo geral, não necessariamente bipartido. Consideraremos grafos sem pesos.

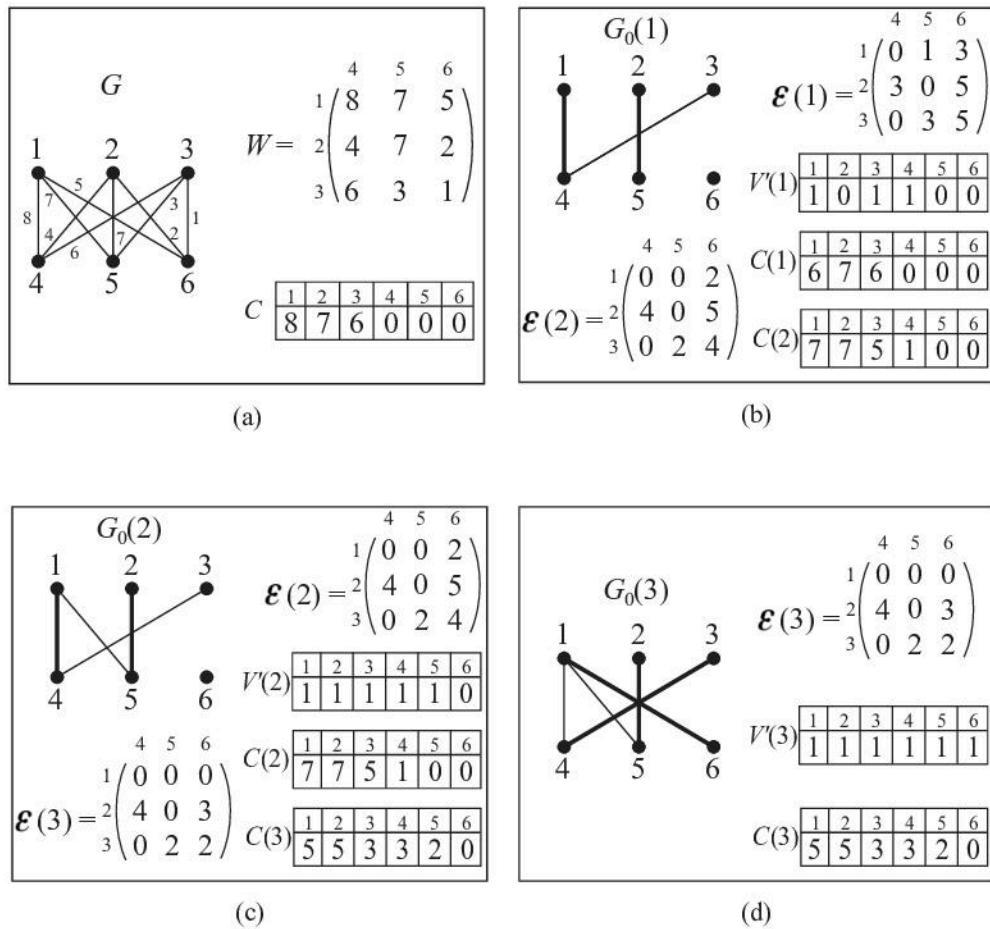


Figura 8.9: Um exemplo de aplicação do Algoritmo 8.3

Número iteração	Vértice escolhido	$c(v_i)$						
		$c(v_1)$	$c(v_2)$	$c(v_3)$	$c(v_4)$	$c(v_5)$	$c(v_6)$	$c(v_7)$
0	$v_1$	0	1	$\infty$	$\infty$	$\infty$	3	$\infty$
1	$v_2$	0	1	$\infty$	$\infty$	$\infty$	3	3
2	$v_7$	0	1	$\infty$	4	$\infty$	3	3
3	$v_6$	0	1	$\infty$	4	8	3	3
4	$v_4$	0	1	$\infty$	4	6	3	3
5	$v_5$	0	1	6	4	6	3	3
6	$v_3$	0	1	6	4	6	3	3

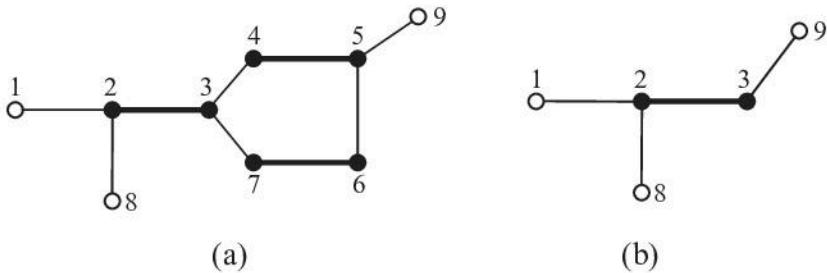


Figura 8.10: Exemplo de emparelhamentos em grafo não bipartido

Inicialmente, observamos que o Teorema 8.2 é válido para qualquer grafo. Portanto, a ideia será utilizada também neste caso, como para grafos bipartidos. Isto é, dado um grafo  $G$  e um emparelhamento  $M$  de  $G$ , qualquer, iterativamente procurar um caminho  $P$  que seja  $M$ -aumentante, incrementar a cardinalidade do emparelhamento através de  $M = M \Delta E_P$ , e assim por diante. Mas, da mesma forma como no caso bipartido, é necessário estabelecer uma estratégia de como encontrar caminhos aumentantes.

Para ilustrar a dificuldade adicional existente nos grafos não bipartidos, veja o grafo e o emparelhamento  $M$  da Figura 8.10(a). Observemos a sequência de vértices  $1, 2, 3, 4, 5, 6, 7, 3, 2, 8$ . As arestas dessa sequência constituem uma sequência alternante, pois em cada par de arestas consecutivas uma pertence a  $M$  e outra não. Além disso, os vértices extremos são  $M$ -expostos. Mas a sequência de vértices não constitui um caminho  $M$ -aumentante, pois possui vértices repetidos, isto é, não é um caminho. Este fenômeno ocorreu devido à presença do ciclo  $3, 4, 5, 6, 7, 3$ , de comprimento ímpar, o que caracteriza um grafo não bipartido, segundo o Teorema 2.2.

O exemplo sugere os conceitos de *passeio  $M$ -alternante* e *passeio  $M$ -aumentante*, similares a caminho  $M$ -alternante e caminho  $M$ -aumentante, exceto que passeios são considerados ao invés de caminhos, isto é, com possível repetição de vértices. Contudo, exigimos que os vértices localizados nas extremidades sejam distintos.

Trataremos, em seguida, deste novo contexto.

Seja  $G$  um grafo, e  $M$  um emparelhamento em  $G$ . Uma  $M$ -floração é um passeio  $M$ -alternante  $v_0, v_1, \dots, v_t$ , tal que  $v_0$  é  $M$ -exposto,  $t$  é ímpar e  $v_t = v_i$ , para algum  $i < t$ . Isto é, consiste em um caminho  $M$ -alternante  $v_0, \dots, v_i$ , de comprimento par denominado *haste*, seguido de um ciclo de comprimento ímpar  $v_i, v_{i+1}, \dots, v_t$ , onde  $v_t = v_i$ , denominado  $M$ -flor. O nó  $v_i$  é a *base* da flor. Observe que  $v_i$  é saturado se e somente se  $i > 0$ .

Considere o exemplo do grafo  $G$  e do emparelhamento  $M$  da Figura 8.10(a). Conforme já observado, a sequência de vértices  $1, 2, 3, 4, 5, 6, 7, 3, 2, 8$  constitui um passeio  $M$ -aumentante, mas não um caminho. A presença do ciclo ímpar  $3, 4, 5, 6, 7, 3$  dá origem a uma  $M$ -floração, cuja haste é o caminho  $1, 2, 3$ , a base é  $3$ , e a  $M$ -flor  $F$  é o ciclo ímpar  $3, 4, 5, 6, 7, 3$ .

Examinamos agora passeios  $M$ -alternantes, com mais detalhe.

Um passeio  $M$ -aumentante é *minimal* quando ele não contém propriamente outro passeio  $M$ -aumentante. O lema seguinte descreve as alternativas deste tipo de passeio.

### Lema 8.5

Seja  $G$  um grafo,  $M$  um emparelhamento de  $G$ , e  $P$  um passeio  $M$ -aumentante em  $G$ , entre  $v_0$  e  $v_t$ . Então ocorre uma das seguintes alternativas:

- (i)  $P$  é um caminho  $M$ -aumentante
- (ii)  $P$  contém uma  $M$ -floração e não contém ciclos simples pares
- (iii)  $P$  não é  $M$ -minimal.

**Prova** Seja  $P$  um passeio  $M$ -aumentante  $v_0, \dots, v_t$  em  $G$ . Se  $P$  for um caminho, (i) se verifica. Suponha que  $P$  não seja um caminho. Então  $P$  contém um ciclo  $C$  de vértices  $w_1, \dots, w_k$ , onde  $w_1 = w_k$ , seguindo a ordem em que  $C$  foi percorrido em  $P$ . Suponha que  $|C|$  seja par. Após percorrer  $C$ , o passeio  $P$  abandona  $C$  através de uma aresta incidente a algum vértice  $w_\ell$  de  $C$ . Podemos verificar que as arestas de  $C$  que estão em  $M$ , são  $(w_1, w_2)$ ,

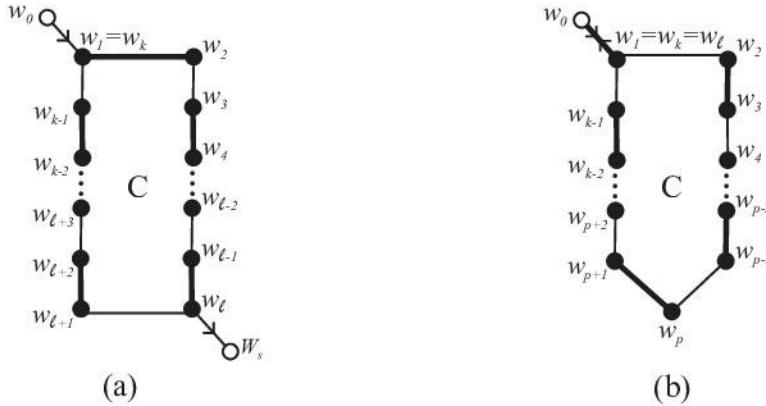


Figura 8.11: Ciclos par (a) e ímpar (b) em passeios

$(w_3, w_4), \dots, (w_{k-2}, w_{k-1})$ , e que além disso  $w_\ell$  é par. Caso contrário, a ordem de alternância das arestas em  $P$ , entre pertencentes ou não pertencentes a  $M$ , não seria obedecida. Os vértices  $w_1, \dots, w_k$  aparecem pelo menos uma vez nesta ordem em  $P$ . Contudo, quando  $P$  abandona  $C$ , apenas os vértices  $w_1, \dots, w_\ell$  são percorridos. Logo, eliminando as arestas  $(w_\ell, w_{\ell+1}), (w_{\ell+1}, w_{\ell+2}), \dots, (w_{k-1}, w_k)$  de  $P$ , obtemos um passeio  $M$ -aumentante propriamente contido em  $P$ . Então  $P$  não é  $M$ -minimal, e (iii) se verifica. A única alternativa restante é  $P$  conter uma  $M$ -floração e não conter ciclos pares, o que equivale à alternativa (ii). ■

As seguintes observações decorrem do Lema 8.5. Seja  $P$  um passeio  $M$ -minimal que não é um caminho. Então,

$P$  contém uma  $M$ -floração e a aresta  $(v, w)$  que conduz ao ciclo ímpar é percorrida exatamente duas vezes em  $P$ , uma vez de  $v$  para  $w$ , e outra no sentido contrário, de  $w$  para  $v$ .

A Figura 8.11(a) e a Figura 8.11(b) ilustram as (únicas) situações possíveis em que ciclos podem aparecer em passeios  $M$ -aumentantes. A Figura 8.11(a) ilustra o caso de ciclo par, e a Figura 8.11(b) o caso de ciclo ímpar. O vértice  $w_1$  é a entrada do passeio no ciclo, e  $w_\ell$  representa a saída em ambas as situações. No caso do ciclo par,  $w_\ell \neq w_1$ , enquanto para o ciclo ímpar,  $w_\ell = w_1$ . Se um ciclo  $C$  for par, o passeio abandona  $C$  através da aresta  $(w_\ell, w_s)$ , onde  $w_\ell \neq w_1$  e  $w_s \notin C$ . Na situação de  $C$  ímpar, o abandono se dá através da aresta  $(w_1, w_0)$ , onde  $w_0 \notin C$ , aresta essa utilizada para entrar em  $C$ . Contudo, para a saída de  $C$ , esta aresta é percorrida em sentido contrário ao utilizado na entrada.

Em seguida, examinamos como tratar as florações. Seja  $F$  uma  $M$ -flor de uma  $M$ -floração de um grafo  $G$ . Represente por  $G/F$ , o grafo obtido pela identificação dos vértices de  $F$ , na base  $b$  da  $M$ -flor, removendo-se as arestas de  $M$  que possuem ambas as extremidades em  $F$ . Nesse caso, observe que a base  $b$  é um vértice  $M/F$ -saturado se e somente se  $b$  for  $M$ -saturado.

O grafo  $G/F$  e o emparelhamento  $M/F$  que se encontram representados na Figura 8.10(b). Observe que a  $M$ -flor 3, 4, 5, 6, 7, 3 de  $G$  foi transformada no único vértice 3, no grafo  $G/F$ . Nesta última, o vértice 3 adquire o vizinho 9, pois este último era vizinho do vértice 5, parte da  $M$ -flor.

Como decorrência do Lema 8.5, as seguintes alternativas ocorrem.

- $G$  não contém passeio  $M$ -aumentante. Neste caso, conclui-se diretamente que  $M$  é máximo.
- O passeio  $M$ -aumentante encontrado é na realidade um caminho  $M$ -aumentante  $P$ . Então podemos aumentar a cardinalidade de  $M$ , através de  $M := M \Delta E_P$ .
- O passeio  $M$ -aumentante encontrado contém uma  $M$ -floração.

A nova situação que se apresenta é a alternativa (iii), iremos examiná-la em seguida, com o auxílio do Lema 8.6.

**Lema 8.6**

Seja  $F$  uma  $M$ -flor de base  $b$ , de uma floração de um grafo  $G$ , e  $v \neq b$  um vértice de  $F$ . Então existe um caminho  $M$ -alternante de comprimento par, entre  $b$  e  $v$ .

**Prova** Há duas possibilidades de percurso em  $F$ , entre  $b$  e  $v$ , ambos correspondem a caminhos  $M$ -alternantes: percorrer  $F$ , entre  $b$  e  $v$ , no sentido horário ou anti-horário. Um desses caminhos possui comprimento par, e outro ímpar. ■

O seguinte teorema é fundamental para resolver o problema do emparelhamento máximo em grafos gerais.

**Teorema 8.5**

(Edmonds) Seja  $G$  um grafo,  $M$  um emparelhamento de  $G$ , e  $F$  uma  $M$ -flor de uma  $M$ -floração de  $G$ . Então  $M$  é máximo em  $G$  se e somente se  $M/F$  é máximo em  $G/F$ .

**Prova** Conforme já observado, a base  $b$  de  $F$  pode ser considerada como não  $M$ -saturada em  $G$ . Além disso, como  $b$  pertence também a  $M/F$ ,  $b$  também não está  $M/F$ -saturado em  $G/F$ . A prova consiste em duas construções, descritas a seguir.

**Construção 1:** Dado um caminho  $M/F$ -aumentante  $P$  em  $G/F$ , obter um caminho  $M$ -aumentante  $P'$  em  $G$ , como a seguir.

Se  $b \notin P$  então basta definir  $P' := P$ . Como  $b$  está  $M/F$ -saturado,  $b$  está também  $M/F$ -saturado e, portanto, não é um dos extremos de  $P$ . Seja  $u$  o vizinho de  $b$  em  $P$ , tal que  $(b, u) \notin M/F$ . Sabemos também que  $u \notin F$ . Então  $u$  possui vizinho  $v$  em  $F$ , cuja identificação com  $b$  resultou na aresta  $(b, u)$ . Obter o caminho  $P'$ , expandindo-se o vértice  $b$  em  $P$ , pelo caminho em  $F$   $M$ -alternante, de comprimento par entre  $b$  e  $v$  (Lema 8.6). O caminho  $P'$  é  $M$ -aumentante.

**Construção 2:** Dado um caminho  $M$ -aumentante  $P'$  em  $G$  obter um caminho  $M/F$ -aumentante  $P$  em  $G/F$ .

Seja  $w_0, w_1, \dots, w_\ell$  um caminho  $M$ -aumentante  $P'$  de  $G$ . Se  $P' \cap (F - \{b\}) = \emptyset$  então basta definir  $P := P'$ . Caso contrário, obtemos  $P$  do seguinte modo. Sabemos que  $w_0, w_\ell \notin F$ , pois  $w_0, w_\ell$  são vértices  $M$ -expostos e todos os vértices de  $F$  não o são. Seja  $w_i$  o vértice de  $P \cap F$  mais próximo a  $w_0$ , e  $w_j \in P \cap F$  o mais próximo a  $w_\ell$ . Então,  $i > 0$  e  $j < \ell$ . Definir  $P$  como sendo a sequência  $w_0, \dots, w_{i-1}, b, w_{j+1}, \dots, w_\ell$ . Consequentemente,  $P$  é um caminho  $M/F$ -aumentante em  $G/F$ .

Como consequência das Construções 1 e 2,  $G$  contém caminho  $M$ -aumentante se e somente se  $G/F$  contém caminho  $G/F$ -aumentante, o que prova o teorema ■

Como exemplo da Construção 1, considere o grafo  $G$  com o emparelhamento  $M$  e a flor  $F$  ilustrados na Figura 8.10(a), bem como o grafo  $G/F$  com o emparelhamento  $M/F$ , da Figura 8.10(b). Seja o caminho  $M/F$ -aumentante  $P$  em  $G/F$ , formado pelos vértices 1, 2, 3, 9. Seguindo a Construção 1, obtemos um caminho  $M$ -aumentante em  $G$ , do seguinte modo. Inicialmente, consideramos a haste  $H$ , de vértices 1, 2, 3 de comprimento par, da floração que contém  $F$ , em  $G$ , bem como a base  $b$ , no caso o vértice 3.

Aplicando, agora, a Construção 1 propriamente dita, e seguindo a notação utilizada, temos:  $b = 3$ ,  $u = 9$ ,  $v = 5$ . O caminho  $Q$ , de comprimento par, entre 3 e 5, em  $F$ , é 3, 4, 5. Então o caminho  $M$ -aumentante  $P'$  é obtido pela substituição do vértice base 3 pelo caminho  $Q$  em  $P'$ . Logo,  $P'$  é o caminho 1, 2, 3, 4, 5, 9.

Como exemplo para a Construção 2, considere novamente o grafo  $G$  e o emparelhamento  $M$ , da Figura 8.10(a). Nesse caso, o vértice base  $b = 3$ . O emparelhamento  $M/F$  aparece na Figura 8.10(b). Considere o caminho  $M$ -aumentante  $P'$  formado pelos vértices 9, 5, 4, 3, 2, 1 na Figura 8.10(a). Sabemos que a  $M$ -flor  $F$  é o ciclo 3, 4, 5, 6, 7, 3. Então  $P' \cap (F - \{b\}) \neq \emptyset$  e, segundo a notação utilizada  $w_0 = 9$ , e  $w_\ell = 1$ . O vértice  $w_i$  pertencente a  $F \cap P'$  mais próximo a  $w_0$  em  $P'$  é 5, e o vértice  $w_j \in F \cap P'$  mais próximo a  $w_\ell$  é 3. Logo, o caminho  $M/F$ -aumentante  $P'$  em  $G/F$ , correspondente a  $P$ , é 9, 3, 2, 1.

No exemplo de aplicação do teorema anterior, considere novamente a Figura 8.10. Concluímos que o emparelhamento  $M$  da Figura 8.10(a) não é máximo, pois o emparelhamento  $M/F$  da Figura 8.10(b) também não é.

A aplicação da estratégia que está sendo apresentada, se baseia no fato de que um passeio  $M$ -alternante é conhecido. Torna-se então necessário desenvolver um método para encontrar passeios  $M$ -aumentantes no grafo, descrito a seguir.

Seja  $G(V, E)$  um grafo qualquer e  $M$  um emparelhamento em  $G$ . Como no caso bipartido, utilizaremos um grafo direcionado auxiliar  $D_f(M)$ . O conjunto de vértices de  $D_f$  é  $V$ , e o conjunto de arestas é definido da seguinte maneira: Para um par de arestas distintas  $(v_1, v_2), (v_2, v_3) \in E$ ,  $(v_1, v_3)$  é aresta direcionada de  $v_1$  para  $v_3$  se e somente se

$$(v_1, v_2) \notin M \text{ e } (v_2, v_3) \in M$$

Assim a existência de uma aresta  $(v_1, v_3)$  em  $D_f$  significa o percurso de um trecho do passeio em construção, no caso da aresta  $(v_1, v_2) \notin M$ , seguida da aresta  $(v_2, v_3) \in M$ . Então, um passeio  $M$ -aumentante corresponde a um caminho  $P_f$  em  $D_f(M)$ , iniciado por algum vértice  $M$ -exposto  $v_0$ , e com término em algum vértice  $M$ -saturado  $v_s$ , tal que  $v_s$  possui vizinho  $v_t \neq v_0$  em  $G$ , onde  $v_t$  é  $M$ -exposto. O vértice  $v_t$  é agregado ao passeio.

O Teorema 8.6 atesta a correção do método.

### Teorema 8.6

Seja  $G$  um grafo e  $M$  um emparelhamento em  $G$ . Então  $G$  contém um passeio  $M$ -aumentante minimal  $P$  se e somente se  $D_f(M)$  contém um caminho direcionado iniciado por algum vértice  $M$ -exposto  $v_0$ , e terminado em um vértice que seja adjacente, em  $G$ , a outro vértice  $M$ -exposto  $v_t \neq v_0$ .

**Prova** Seja  $P$  um passeio  $M$ -aumentante minimal  $v_0, v_1, \dots, v_{t-1}, v_t$  em  $G$ . Como as arestas  $(v_i, v_{i+1})$ , alternadamente, não pertencem e pertencem a  $M$ , e os vértices  $v_0, v_t$  são  $M$ -expostos, então as arestas  $(v_0, v_1), (v_2, v_3), \dots, (v_{t-1}, v_t) \notin M$ , e as demais arestas pertencem a  $M$ . Em consequência, o grafo  $D_f(M)$  contém as arestas direcionadas  $(v_0, v_2), (v_2, v_4), \dots, (v_{t-3}, v_{t-1})$ , onde  $v_{t-1}$  é adjacente, em  $G$ , a  $v_t$ . Resta mostrar que os vértices  $v_0, v_2, \dots, v_{t-3}, v_{t-1}$  são todos distintos. Se  $P$  é um caminho em  $G$ , então todos os vértices  $v_0, v_1, \dots, v_{t-1}, v_t$  são distintos. Caso contrário, como  $P$  é minimal,  $P$  não contém ciclos pares. Por outro lado, sabemos que se  $P$  contém um ciclo ímpar  $C$ , de vértices,  $w_1, w_2, \dots, w_{k-1}, w_k$ , onde  $w_k = w_1$ , então as arestas de  $D_f$  correspondentes ao percurso de  $C$  em  $D_f$  são  $(w_1, w_3), (w_3, w_5), \dots, (w_{k-1}, w_0)$ , onde  $(w_0, w_1) \in M$  e  $w_0 \notin C$ . Logo,  $C$  é percorrido exatamente uma vez em  $P$ . Consequentemente, os vértices  $v_0, v_2, \dots, v_{t-3}, v_{t-1}$  são todos distintos, o que implica que  $(v_0, v_2), (v_2, v_4), \dots, (v_{t-3}, v_{t-1})$  são as arestas de um caminho em  $D_f$ , iniciado pelo vértice  $M$ -exposto  $v_0$ , e terminado pelo vértice  $v_{t-1}$ , adjacente em  $G$ , ao vértice  $M$ -exposto  $v_t \neq v_0$ .

Reciprocamente, sejam  $(v_0, v_2), (v_2, v_4), \dots, (v_{t-3}, v_{t-1})$  as arestas de um caminho  $P_f$ , em  $D_f$ , tal que  $v_0$  é  $M$ -exposto,  $v_{t-1}$  é adjacente, em  $G$ , a um vértice  $v_t$  também  $M$ -exposto,  $v_t \neq v_0$ . Obter um passeio  $M$ -aumentante minimal  $P$ , correspondente a  $P_f$ , como se segue:

**Construção 3:** Escolher vértices  $v_1, v_3, \dots, v_{t-2} \in V$ , tal que

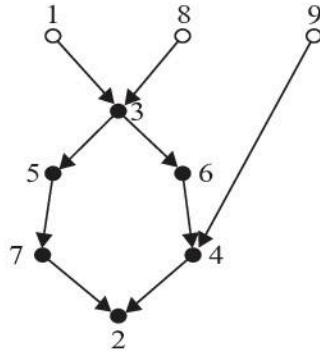
$$(v_0, v_1), (v_2, v_3), \dots, (v_{t-3}, v_{t-2}) \in E - M, \text{ e}$$

$$(v_1, v_2), (v_3, v_4), \dots, (v_{t-2}, v_{t-1}) \in M.$$

Construir  $P$  como sendo o passeio  $v_0, v_1, v_2, \dots, v_{t-1}, v_t$ .

A escolha dos vértices  $v_1, v_3, \dots, v_{t-2} \in V$ , satisfazendo as condições descritas na Construção 3, é sempre possível, pois decorre diretamente da definição do grafo  $D_f(M)$ . Cada par de vértices consecutivos de  $P$  corresponde a uma aresta de  $G$ . Logo,  $P$  é um passeio. Além disso, em cada par de arestas consecutivas de  $P$ , uma pertence a  $M$  e outra não pertence.

Como  $v_0, v_t$  são vértices  $M$ -expostos, o passeio é  $M$ -aumentante. Resta mostrar que  $P$  é minimal. Suponha o contrário. Logo  $P$  contém um ciclo par, de vértices  $w_1, w_2, \dots, w_{k-1}, w_k$ , onde  $w_k = w_1$ , e  $k$  é ímpar. Por outro lado, sabemos que  $(w_1, w_2), (w_3, w_4), \dots, (w_{k-2}, w_{k-1}) \notin M$ . Este ciclo corresponderia então em  $D_f$

Figura 8.12: Grafo direcionado  $D_f(M)$  da Figura 8.10(a)

ao percurso  $(w_1, w_3), (w_3, w_5), \dots, (w_{k-2}, w_k), (w_1, w_3), \dots$ . Logo, este percurso em  $D_f$  conteria vértices repetidos, o que contradiz o mesmo ser um caminho. Logo,  $P$  não pode conter ciclos pares, consequentemente,  $P$  é minimal. ■

Como exemplo da Construção 3, utilizamos o caminho inverso do exemplo acima, referente ao grafo  $G$  e emparelhamento  $M$  da Figura 8.10(a). No caso, supomos que seja dado o caminho  $P_f$ , em  $D_f$ , formado pelos vértices 1, 3, 5, 7, 2, iniciado pelo vértice  $M$ -exposto 1, e terminado pelo vértice  $M$ -saturado 2, tal que  $(2, 8) \in E$ , e 8 é  $M$ -exposto. De acordo com a notação da Construção 3,  $v_0 = 1$ ,  $v_{t-1} = 2$ ,  $v_t = 8$ , o que conduz a  $v_0 = 1$ ,  $v_2 = 3$ ,  $v_4 = 5$ ,  $v_6 = 7$ ,  $v_8 = 2$  e  $v_9 = 8$ , com  $t = 9$ . A escolha de vértices  $v_1, v_3, \dots, v_{t-2}$ , satisfazendo às condições da Construção 3, conduziria a:  $v_1 = 2, v_3 = 4, v_5 = 6, v_7 = 3$ . Assim, o passeio  $M$ -aumentante  $P$  em  $G$ , correspondente a  $P_f$ , formado pelos vértices  $v_0, v_1, v_2, \dots, v_{t-1}, v_t$ , no caso é igual a 1, 2, 3, 4, 5, 6, 7, 3, 2, 8.

Considere o grafo  $G$  e o emparelhamento  $M$  da Figura 8.10(a). Na Figura 8.12 aparece o grafo direcionado  $D_f(M)$  correspondente ao grafo e emparelhamento representados na Figura 8.10(a). O passeio  $M$ -aumentante 1, 2, 3, 7, 6, 5, 4, 3, 2, 8, corresponde ao caminho 1, 3, 6, 4, 2 em  $D_f$ , iniciado pelo vértice  $M$ -exposto 1 e terminado pelo vértice  $M$ -saturado 2, o qual possui o vizinho  $M$ -exposto  $8 \neq 1$ .

O algoritmo agora está aparente. Dado um grafo  $G$  e um emparelhamento  $M$ , possivelmente vazio, construir o grafo  $D_f(M)$ , bem como um caminho em  $D_f$ , iniciado em um vértice  $v_0$   $M$ -exposto, e terminado em um vértice  $v_s$  que possua um vizinho  $v_t$  em  $G$ ,  $M$ -exposto. Se  $D_f$  não contém tal caminho então  $M$  é máximo (Teorema 8.6), e o algoritmo termina. Caso contrário, construir o passeio  $M$ -aumentante minimal em  $G$  (Construção 3). Se  $P$  for um caminho  $M$ -aumentante, então aumentar a cardinalidade de  $M$ , através de  $M := M \Delta E_P$  (Teorema 8.2), onde  $E_P$  representa o conjunto das arestas que formam  $P$ . Caso contrário,  $P$  contém uma  $M$ -floração, e denote por  $F$  a  $M$ -flor correspondente. Construir o grafo  $G/F$  e o emparelhamento  $M/F$ , e repetir o processo para  $G/F$  e  $M/F$ , recursivamente, para verificar se existe caminho  $P/F$  que seja  $M/F$ -aumentante em  $G/F$ . Se não existir tal caminho, então  $M/F$  é máximo em  $G/F$ , pelo Teorema 8.5  $M$  é máximo em  $G$ , e o algoritmo termina. Caso contrário, transformar o caminho  $P/F$ -aumentante de  $G/F$ , em um caminho  $P$  que seja  $M$ -aumentante em  $G$ , de acordo com a Construção 1 do Teorema 8.5. Construir  $M := M \Delta E_P$ , e repetir o processo para  $G$  e o valor atualizado de  $M$ .

O Algoritmo 8.4 descreve o processo. A sua entrada é um grafo  $G(V, E)$ , conexo, e a saída é um emparelhamento máximo em  $G$ . Ele utiliza a Construção 1 do Teorema 8.5, a qual constrói um caminho  $M$ -aumentante  $P$ , correspondente ao caminho  $M/F$ -aumentante do grafo  $G/F$ , para a  $M$ -flor selecionada  $F$  de  $G$ .

O Algoritmo 8.4 utiliza o procedimento AUMENTANTE ( $G, M, P$ ). Para o grafo  $G$  e um emparelhamento  $M$  de  $G$ , o procedimento verifica se  $G$  contém passeio  $M$ -aumentante  $P$ . Com este propósito, inicialmente, o procedimento constrói o grafo auxiliar  $D_f(M)$ . Em  $D_f$ , o objetivo é construir um caminho  $P_f$  de  $v_0$  a  $v_s$ , tal que,  $v_0, v_t$  sejam vértices  $M$ -expostos e  $(v_s, v_t) \in E$ . Se  $D_f$  contiver um caminho  $P_f$  com estas propriedades, então  $P_f$  é transformado em um caminho  $M$ -aumentante  $P$  de  $G$ , através da Construção 3. Caso contrário, define-se  $P := \emptyset$ . O procedimento retorna a informação sobre o caminho  $P$ , a qual é utilizada pelo Algoritmo 8.4.

O procedimento AUMENTANTE, por sua vez, utiliza o procedimento REDUÇÃO-FLOR( $G', M', P'$ ), o qual, recursivamente, irá construindo a  $M'$ -flor de  $G'$ , cada vez que esta é encontrada. O passeio  $P'$  de  $G'$  é então transformado no passeio  $P$  de  $G$ , através da Construção 1. As construções 1 e 3 correspondem também a procedimentos que são chamados pelo algoritmo principal, e pelos procedimentos AUMENTANTE e REDUÇÃO-FLOR. Isto é, CONSTRUÇÃO\_1( $G', M', P', M, G$ ) é uma função, que associa o caminho  $M$ -aumentante do grafo  $G$ , ao caminho  $M'$ -aumentante do grafo  $G'$ , onde  $G' = G/F$  e  $M' = M/F$ , para uma certa flor  $F$ .

O algoritmo principal, Algoritmo 8.4, bem como os procedimentos AUMENTANTE e REDUÇÃO-FLOR são detalhados a seguir. Com isso, completamos a descrição do algoritmo para determinação do emparelhamento máximo em grafos gerais.

---

**Algoritmo 8.4:** Emparelhamento em grafos gerais

---

**Dados:** Grafo  $G(V, E)$

**repetir**

    REDUÇÃO-FLOR( $G, M, P$ )

$M := M \Delta E_p$

**até que**  $P = \emptyset$

---



---

**Procedimento** REDUÇÃO-FLOR( $G, M, P$ )

---

    AUMENTANTE( $G, M, P$ )

**se**  $P$  contém  $M$ -flor  $F$  **então**

$G' := G/F; M' := M/F$

        REDUÇÃO-FLOR( $G', M', P'$ )

$P := \text{CONSTRUÇÃO}_1(G', M', P', M, G)$

---



---

**Procedimento** AUMENTANTE( $G, M, P$ )

---

$D_f(M) :=$  grafo direcionado auxiliar

**se** existir caminho  $P_f$  de  $v_0$  a  $v_s$  em  $D_f$ , tal que  $(v_s, v_t) \in E$ , e  $v_0, v_t$  são  $M$ -expostos **então**

$P := \text{CONSTRUÇÃO}_3(P_f, D_f, G, M)$

---

A complexidade do Algoritmo 8.4 pode ser determinada da seguinte maneira. A construção do grafo  $D_f(M)$ , no procedimento AUMENTANTE, pode ser realizada em  $O(m)$  passos. A busca do caminho  $P_f$ , bem como a sua transformação em um passeio  $M$ -aumentante minimal de  $G$  requerem  $O(m)$  passos. Assim, o procedimento AUMENTANTE termina em  $O(m)$  passos. Cada chamada do procedimento REDUÇÃO-FLOR, a partir do algoritmo principal, termina em  $O(n(n+m))$  passos, incluindo as chamadas recursivas por ele geradas. Pois, cada redução diminui o emparelhamento em pelo menos dois vértices, e no máximo  $O(n)$  chamadas do procedimento são realizadas. A complexidade final é, pois,  $O(n^2m)$ .

## 8.9 Programas em Python

Esta seção contém implementações dos algoritmos formulados neste capítulo. As implementações seguem as descrições gerais apresentadas nos Capítulos 1 e 2.

### 8.9.1 Algoritmo 8.1: Emparelhamento Bipartido (com Digrafo)

A implementação do Algoritmo 8.1 é uma tradução direta para a linguagem, observando-se que as Linhas 7–10 implementam a condição de parada do algoritmo.

## Programa 8.1: Emparelhamento cardinalidade máxima - Grafos bipartidos

```

#Algoritmo 8.1: Emparelhamento cardinalidade máxima - grafos bipartidos
#Dado: Grafo bipartido G com bipartição V1 U V2, onde V1 = {1,...,G.tV1}; V2 = ↓
{G.tV1+1,...,|V(G)|}
#      Retorna emparelhamento M[1..n], onde: (i) M[v] = None <==> v é exposto; (ii) M[v] = w ↓
<==> vw in M

def EmparelhamentoBipartido(G):
    M = [None]*(G.n+1)
    while True:
        D = ObterD(G, M)
        P = BuscaCamAumentante(D)
        if len(P) == 0:
            break
        M = DifSimetrica(M, P)
    return M

def ObterD(G, M):
    """ G: Grafo, M: emparelhamento. Retorna D(M). """
    D = GrafoListaAdj(orientado = True)
    D.DefinirN(G.n+1)
    D.tV1 = G.tV1
    D.Exp = [False]*(D.n+1)
    for v in range(1, D.n):
        D.Exp[v] = (M[v] == None)
    for v in range(1, G.tV1+1):
        if D.Exp[v]:
            D.AgregarAresta(D.n, v)
        for w in G.N(v):
            if M[v] == w:
                D.AgregarAresta(w, v)
            else:
                D.AgregarAresta(v, w)

    return D

def BuscaCamAumentante(D):
    """ D: digrafo com grafo subjacente conexo. Retorna (as arestas de) um caminho ↓
        aumentante, se existente. """
    def P(v):
        D.Marcado[v] = True
        Q.append(v)
        if D.tV1 < v < D.n and D.Exp[v]:
            return True
        for w in D.N(v, "+"):
            if not D.Marcado[w]:
                if P(w):
                    return True
        Q.pop()
        return False
    D.Marcado = [False]*(D.n+1);
    Q = []
    s = D.n
    P(s)
    return Q[1:]

def DifSimetrica(M, P):

```

```

53     """ M: emparelhamento, P: (arestas de) um caminho. Retorna a diferença simétrica de M por ↓
54     P """
55     MR = [ v for v in M ]
56     for i in range(0, len(P), 2):
57         MR[P[i]], MR[P[i+1]] = P[i+1], P[i]
58     return MR

```

### 8.9.2 Algoritmo 8.2: Emparelhamento Bipartido (Método Húngaro)

Na implementação do Algoritmo 8.2, o conjunto EXPOSTO foi representado por uma tabela de acesso direto EXPOSTO, tal que  $\text{EXPOSTO}[v] = \text{True}$  se e somente se  $v \in \text{EXPOSTO}$ . A estratégia da implementação do comando de repetição é ligeiramente modificada, seguindo a análise da complexidade de tempo (final da Seção 8.5). Na implementação, itera-se sobre cada aresta  $(u, v)$ . Quando achamos uma aresta em que  $u$  seja par e  $v$  não seja ímpar, executa-se o corpo da repetição. Quando  $M$  é aumentado, reiniciamos a iteração sobre o conjunto das arestas de  $G$ . A diferença simétrica é feita tal como no Algoritmo 8.1.

Programa 8.2: Emparelhamento cardinalidade máxima - Método Húngaro

```

#Algoritmo 8.2: Emparelhamento cardinalidade máxima - Método Húngaro
2 #Dado: Grafo bipartido G com bipartição V1 U V2, onde V1 = {1, ..., G.tV1}; V2 = ↓
   {G.tV1+1, ..., /V(G)/}
3 def EmparelhamentoBipartidoMetodoHungaro(G):
4     #uv in M <==> M[u] = v e M[v] = u
5     M = [None]*(G.n+1); EXPOSTO = [True]*(G.n+1)
6     #T[v] é pai de v na arvore de T ou T[v] = v se v é raiz de sua arvore
7     T = [None] + [v for v in G.V()]
8     #par[v] == True <==> v é par(F); ímpar[v] == True <==> v é ímpar(F)
9     par = [True]*(G.n+1); impar = [False]*(G.n+1)
10
11    E = G.E(); (u,v) = next(E, (None, None))
12    while u != None:
13        if par[u] and not impar[v]:
14            if not par[v]:
15                #AUMENTAR(F)
16                vlin = M[v]
17                T[v], T[vlin] = u, v
18                par[vlin] = True; impar[v] = True
19            else:
20                #AUMENTAR(M)
21                r = u
22                P = [r]
23                while T[r] != r:
24                    P.append(T[r])
25                    r = T[r]
26                P.reverse()
27                rlin = v
28                P.append(rlin)
29                while T[rlin] != rlin:
30                    P.append(T[rlin])
31                    rlin = T[rlin]
32
33                M = DifSimetrica(M, P)
34
35                EXPOSTO[r] = False; EXPOSTO[rlin] = False
36                par = [None] + [ EXPOSTO[v] for v in G.V() ]
37                impar = [False]*(G.n+1)

```

```

38     T = [None] + [ (v if EXPOSTO[v] else None) for v in G.V() ]
39     E = G.E()
40     (u,v) = next(E, (None, None))
41     return M

```

### 8.9.3 Algoritmo 8.3: Emparelhamento Bipartido Ponderado

O Algoritmo 8.3 é implementado como tradução direta para a linguagem. As funções EmparelhamentoBipartido, ObterD e BuscaCamAumentante são aquelas definidas no Algoritmo 8.1.

Programa 8.3: Emparelhamento ponderado máximo - Grafos bipartidos

```

#Algoritmo 8.3: Emparelhamento ponderado máximo - Grafos bipartidos
#Dados: grafo bipartido completo regular ponderado G, peso real w[v1][v2] >= 0 para cada ↓
#aresta (v1,v2) em E
def EmparelhamentoBipartidoPonderado(G, w):
    c = [None]*(G.n+1)
    e = [None]*(G.n+1)
    for i in range(1,G.n+1):
        e[i] = [None]*(G.n+1)
    G.tV1 = G.n//2

    def DefinirGO(G,e):
        G0 = GrafoListaAdj(orientado = False)
        G0.DefinirN(G.n)
        G0.tV1 = G0.n//2
        G0.d = [0]*(G0.n+1)
        E0 = [(v1,v2) for (v1,v2) in G.E() if e[v1][v2] == 0.0]
        for (v1,v2) in E0:
            G0.AdicionarAresta(v1,v2)
            G0.d[v1] += 1; G0.d[v2] += 1
        return G0

    for v in range(1,G.tV1+1):
        c[v] = max ([w[v][v2] for v2 in G.N(v)])
    for v in range(G.tV1+1,G.n+1):
        c[v] = 0
    for (v1,v2) in G.E():
        e[v1][v2] = c[v1] + c[v2] - w[v1][v2]; e[v2][v1] = e[v1][v2]
    G0 = DefinirGO(G,e)
    while True:
        M = EmparelhamentoBipartido(G0)
        if len([i for i in range(len(M)) if M[i] != None]) < G.n:
            D = ObterD(G0,M)
            BuscaCamAumentante(D) #D.Marcado[v] <=> v in Vlin
            eps = [e[v1][v2] for v2 in range(G.tV1+1,G.n+1) for v1 in range(1,G.tV1+1) if (not ↓
                D.Marcado[v2]) and D.Marcado[v1]]
            emin = min(eps)
            for v in range(G.tV1+1,G.n+1):
                if D.Marcado[v] > 0:
                    c[v] = c[v] + emin
            for v in range(1,G.tV1+1):
                if D.Marcado[v] > 0:
                    c[v] = c[v] - emin

```

```

41     for (v1,v2) in G.E():
42         e[v1][v2] = c[v1] + c[v2] - w[v1][v2]; e[v2][v1] = e[v1][v2]
43         G0 = DefinirG0(G,e)
44     else:
45         break
46 return (M, c)

```

### 8.9.4 Algoritmo 8.4: Emparelhamento Geral

As funções do Algoritmo 8.4 seguem em maior parte de forma direta, com pequenas adaptações para a linguagem Python (como a mudança dos parâmetros passados por referência no algoritmo para uma devolução de função em Python).

A função DifSimetrica é aquela do Algoritmo 8.1 e, portanto, encontra-se omitida. As Construções 1 e 3 citadas no algoritmo são detalhadas nas funções CONSTRUCAO\_1 e CONSTRUCAO\_3, respectivamente.

Programa 8.4: Emparelhamento em grafos gerais

```

#Algoritmo 8.4: Emparelhamento em grafos gerais
#Dado: Grafo G
def EmparelhamentoGeral(G):
    #uv in M <==> M[u] = v e M[v] = u
    M = [None]*(G.n+1)

    while True:
        P = ReducaoFlor(G, M)
        M = DifSimetrica(M, P)
        if len(P) == 0:
            break
    return M

def ReducaoFlor(G, M):
    """ GF: Grafo reduzido de uma flor pela CONSTRUCAO 1 """
    (P, F, H) = Aumentante(G, M)
    if len(P) > 0 and F != None:
        (GF, MF) = ObterGFMF(G, M, F, H)
        Plin = ReducaoFlor(GF, MF)
        P = CONSTRUCAO_1(G, GF, F, H, M, Plin)
    return P

def Aumentante(G, M):
    #Construcao de Df
    Df = GrafoListaAdj(orientado=True)
    Df.DefinirN(G.n)
    Df.ExpAssoc = [None]*(Df.n+1)
    for v in G.V():
        for w in G.N(v):
            if M[w] != v: #vw nao esta em M
                if M[w] == None:
                    Df.ExpAssoc[v] = w
                else:
                    e = Df.AdicionarAresta(v, M[w])
                    e.inter = w
    Df.Exp = [True]*(Df.n+1)
    for v in M:

```

```

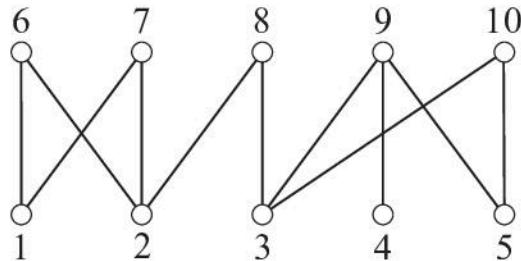
40     if v != None:
41         Df.Exp[v] = False
42
43 Pf = BuscaCamAumentante(Df)
44
45 (P, F, H) = CONSTRUCAO_3(Pf, Df, G, M)
46
47 return (P, F, H)
48
49 def ObterGFMF(G, M, F, H):
50     GF = GrafoListaAdj(orientado=False)
51     GF.DefinirN(G.n-len(F)+1)
52     GF.VAssocD = [None]*(GF.n+1)
53     G.VAssocO = [None]*(G.n+1)
54     G.VizEmF = [None]*(G.n+1)
55
56 ArestaComFlor = [False]*(G.n+1)
57
58 #vértices de F em G serao associados ao vértice 1 de GF
59 for v in F:
60     G.VAssocO[v] = 1
61     GF.VAssocD[1] = F[0]
62 n = 2
63 for v in G.V():
64     if G.VAssocO[v] == None:
65         G.VAssocO[v] = n
66         GF.VAssocD[n] = v
67     n=n+1
68
69 MF = [None]*(GF.n+1)
70 for v in G.V():
71     if M[v] != None and G.VAssocO[v] != G.VAssocO[M[v]]:
72         MF[G.VAssocO[v]] = G.VAssocO[M[v]]
73
74 for v in G.V():
75     for w in G.N(v):
76         if v < w:
77             (x, y) = (v, w) if G.VAssocO[v] == 1 else (w, v)
78             if G.VAssocO[y] != 1:
79                 if G.VAssocO[x] == 1:
80                     if G.VizEmF[y] == None or GF.VAssocD[1] == x:
81                         G.VizEmF[y] = x
82                         if not ArestaComFlor[y]:
83                             ArestaComFlor[y] = True
84                             GF.AdicionarAresta(G.VAssocO[x], G.VAssocO[y])
85                 else:
86                     GF.AdicionarAresta(G.VAssocO[x], G.VAssocO[y])
87
88 return (GF, MF)
89
90 def CONSTRUCAO_1(G, GF, F, H, M, PF):
91     if len(PF) == 0:
92         return []
93
94 P = []
95 if PF.count(1)>0:

```

```

96     indice = PF.index(1)
97     if not G.EhAresta(GF.VAssocD[1],GF.VAssocD[PF[indice+1]]):
98         PF.reverse()
99
100    for i in range(len(PF)):
101        if PF[i] != 1:
102            P.append(GF.VAssocD[PF[i]])
103        else:
104            #colocar em P o caminho base ate saída da flor de tamanho par
105            indice = F.index(G.VizEmF[GF.VAssocD[PF[i-1]]])
106            if indice % 2 == 0:
107                for j in range(indice, 0, -1):
108                    P.append(F[j])
109            else:
110                for j in range(indice, len(F)):
111                    P.append(F[j])
112            P.append(F[0])
113
114    return P
115
116 def BuscaCamAumentante(D):
117     """ D: digrafo """
118     def P(v):
119         D.Marcado[v] = True
120         Q.append(v)
121         if D.ExpAssoc[v] != None and (not D.Marcado[D.ExpAssoc[v]]):
122             Q.append(D.ExpAssoc[v])
123             return True
124         for w_no in D.N(v, "+", IterarSobreNo=True):
125             w = w_no.Viz
126             if not D.Marcado[w] and (not D.Marcado[w_no.e.inter] or not w_no.e.inter in Q):
127                 Q.append(w_no.e.inter)
128                 if P(w):
129                     return True
130                 Q.pop()
131             Q.pop()
132             return False
133     Q = []
134     for s in D.V():
135         if D.Exp[s]:
136             D.Marcado = [False]*(D.n+1)
137             if P(s):
138                 break
139     return Q
140
141 def CONSTRUCAO_3(Pf, Df, G, M):
142     Ciclo = None; H = None
143     VMarcado = [-1]*(Df.n+1)
144     i = 0
145     for v in Pf:
146         if VMarcado[v] > -1:
147             Ciclo = Pf[VMarcado[v]:i]
148             H = Pf[:VMarcado[v]+1]
149             break
150     else:
151         VMarcado[v] = i; i=i+1

```

Figura 8.13: Grafo  $G$ 

152

**return** (Pf, Ciclo, H)

## 8.10 Exercícios

- 8.1 Mostrar que uma árvore admite, no máximo, um emparelhamento perfeito
- 8.2 Deducir as condições necessárias e suficientes para que uma árvore admita emparelhamento perfeito.
- 8.3 Seja  $G$  um grafo bipartido e  $M$  um emparelhamento em  $G$ . Descrever os algoritmos para realizar os seguintes passos contidos no Algoritmo 8.1, que determinam um emparelhamento máximo em um grafo bipartido conexo  $G(V, E)$ .
- (i) Construir o grafo  $D(M)$
  - (ii) Encontrar um caminho  $P$ , em  $D(M)$ , entre vértices não  $M$ -saturados, localizados em partes distintas de  $D(M)$ .
- 8.4 Aplicando o Algoritmo 8.1 no grafo  $G$  da Figura 8.13, descrever os passos que o algoritmo efetuará, até encontrar um emparelhamento máximo de  $G$ . Escolher, arbitrariamente, uma alternativa possível, compatível com o algoritmo.
- 8.5 Escrever um algoritmo para encontrar uma cobertura mínima das arestas por vértices, em um grafo bipartido.
- 8.6 Um *emparelhamento induzido*  $M$  em um grafo  $G$  é um emparelhamento tal que  $(v, w) \in M \Rightarrow d(v, u) > 1$ , para todo vértice  $M$ -emparelhado  $u \neq v, w$ . Formular um algoritmo para determinar um emparelhamento induzido máximo em uma árvore.
- 8.7 Dado um grafo bipartido  $G(V, E)$ , descrever como obter um emparelhamento máximo em  $G$ , através da solução de um problema de fluxo máximo em redes. Descrever a rede correspondente.
- 8.8 Seja um tabuleiro  $8 \times 8$  de onde foram removidos dois cantos opostos, de tamanho  $1 \times 1$ . Mostrar que é impossível cobrir exatamente este tabuleiro utilizando retângulos  $2 \times 1$ .

- 8.9 É possível generalizar o exercício anterior, para um tabuleiro  $(2k) \times (2k)$ ,  $k \geq 1$ ? O que ocorre com tabuleiros  $(2k+1) \times (2k+1)$ ?
- 8.10 Seja um grafo  $G(V, E)$ , e  $V' \subseteq V$ . Seja  $\Phi(E, V')$  o número de componentes conexas de  $G - V'$ , com um número ímpar de vértices. Provar a seguinte caracterização:
- $$G \text{ possui emparelhamento perfeito se e somente se } \Phi(G, V') \leq |V'|.$$
- 8.11 Um grafo bipartido  $G(V, E)$ , com bipartição  $V = W \cup U$ , é dito *convexo* se uma das partes, digamos  $W$ , admitir uma ordenação  $w_1, \dots, w_{|w|}$  de vértices, tal que para  $j < k$ ,
- $$(w_j, u), (w_k, u) \in E \implies (w_{j+1}, u), \dots, (w_{k-1}, u) \in E,$$
- para qualquer  $u \in U$ . Formular um algoritmo guloso para encontrar o emparelhamento máximo nessa classe de grafos. Determinar a sua complexidade.
- 8.12 Provar ou dar contra-exemplo:
- Considere o Algoritmo 8.2, para a determinação do emparelhamento máximo ponderado de um grafo bipartido regular. Então, em cada iteração do bloco **repetir** do algoritmo, a cardinalidade do emparelhamento máximo  $M$  de  $G_0$  aumenta.
- 8.13 Apresentar um exemplo de um grafo bipartido completo regular ponderado com exatamente uma aresta de peso máximo, e tal que esta aresta não possa ser incluída no emparelhamento máximo ponderado do grafo.
- 8.14 Seja  $G$  um grafo,  $M$  um emparelhamento de  $G$ , e  $F$  uma  $M$ -flor de  $G$ . Descrever, em detalhes os algoritmos a seguir (i) e (ii), que implementam as Construções 1 e 2, da prova do Teorema 8.5.
- (i) Seja  $P$  um caminho  $M/F$ -aumentante de  $G/F$ . Construir um caminho  $M$ -aumentante  $P'$  de  $G$ , em função de  $P$ .
  - (ii) Seja  $P'$  um caminho  $M$ -aumentante de  $G$ . Construir um caminho  $M/F$ -aumentante de  $G/F$ , em função de  $P'$ .
- Determine a complexidade dos algoritmos descritos.
- 8.15 Seja  $G$  um grafo,  $M$  um emparelhamento de  $G$ ,  $F$  uma  $M$ -flor de  $G$ . Escrever, em detalhe, os algoritmos a seguir, que implementam as construções do Teorema 8.6
- (i) Construir o grafo direcionado  $D_f(M)$
  - (ii) Construir um caminho  $P_f$  em  $D_f$ , iniciado por um vértice  $M$ -exposto  $v_0$ , e terminado por um vértice  $v_{t-1}$ , que seja adjacente em  $G$ , a um vértice  $M$ -exposto  $v_t \neq v_0$
  - (iii) Construir o passeio minimal  $P$ , em  $G$ , correspondente ao caminho  $P_f$ , obtido em (ii), de acordo com a Construção 3.
- 8.16 Seja  $P$  um passeio  $M$ -aumentante em  $G$ , contendo um ciclo  $C$ , de vértices  $w_1, \dots, w_{k-1}, w_k$ , onde  $w_k = w_1$ ,  $k \geq 4$ . Seja  $(w_0, w_1)$  a aresta de entrada em  $C$ , no percurso de  $P$ , e  $(w_\ell, w_{\ell+1})$  a aresta por onde  $P$  abandona  $C$ . Mostrar que

- (i) Se  $C$  é par então  $(w_0, w_1), (w_2, w_3), \dots, (w_{k-1}, w_k), (w_\ell, w_{\ell+1}) \notin M$ ,
- (ii) Se  $C$  é ímpar então  $(w_0, w_1) \in M$  e  $w_\ell = w_1$ .

8.17 Seja  $G$  um grafo qualquer. Um certo jogo, em  $G$ , para dois jogadores, possui as seguintes regras. Em cada jogada, os jogadores, alternadamente, escolhem vértices  $v_0, v_1, v_2, \dots$ , de modo que, na jogada  $i > 0$ , o jogador escolhe um vértice  $v_i$ , ainda não previamente escolhido, tal que  $v_i$  é adjacente ao vértice  $v_{i-1}$ . Mostrar que o jogador que efetuou a jogada inicial possui uma estratégia vencedora se e somente se  $G$  não possui emparelhamento perfeito. Descrever a estratégia vencedora.

8.18 Seja  $G(V, E)$  um grafo,  $U \subseteq V$  e  $\Phi(U)$  o número de componentes conexas do grafo  $G - U$ . Provar:  $G$  possui um emparelhamento perfeito se e somente se  $\Phi(U) \leq |U|$ , para cada  $U \subseteq V$  (Teorema de Tutte).

8.19 Mostrar que qualquer grafo 3-regular e 2-conexo em arestas possui um emparelhamento perfeito (Teorema de Petersen).

8.20 Mostrar que todo grafo bipartido  $k$ -conexo possui um emparelhamento perfeito.

8.21 Seja  $G(V, E)$  um grafo, e  $S \subseteq V$  tal que existe um emparelhamento  $M$  de  $G$ , em que todos os vértices de  $S$  são  $M$ -emparelhados. Provar ou dar contra-exemplo: Existe um emparelhamento máximo  $M'$  de  $G$ , em que os vértices de  $S$  são  $M'$ -emparelhados.

8.22 Seja  $G(V, E)$  um grafo bipartido completo ponderado, isto é, em que cada aresta  $e \in E$  está associada a um real não negativo, denominado peso de  $e$ . Seja  $M$  um emparelhamento de  $G$ . Definir o peso de  $M$ , como a soma dos pesos das arestas que formam  $M$ . Descrever um algoritmo para determinar um emparelhamento perfeito de  $G$ , de peso mínimo.

8.23 Seja  $G(V, E)$  um grafo bipartido ponderado. Descrever um algoritmo para determinar um emparelhamento de  $G$ , de peso máximo.

8.24 UVA Online Judge 11419

Escreva um algoritmo para o seguinte problema:

João está jogando um jogo onde tem que destruir inimigos em um templo. O templo tem o formato de uma grade e os inimigos estão espalhados nas células da grade. João tem que atirar balas de canhão ou nas linhas ou nas colunas, que são capazes de eliminar todos os inimigos na linha do tiro. São dadas as dimensões  $a$  e  $b$  da grade, o número  $n$  de inimigos e as  $n$  posições dos inimigos na grade. Calcular o número mínimo de tiros para eliminar todos os inimigos.

8.25 UVA Online Judge 11159

Escreva um algoritmo para o seguinte problema:

Dados dois conjuntos de inteiros, determinar o número mínimo de elementos que devem ser retirados dos dois conjuntos tal que nenhum inteiro de um conjunto seja múltiplo de qualquer inteiro do outro.

## 8.11 Notas Bibliográficas

Emparelhamentos constituem um dos tópicos mais relevantes em teoria de grafos, tanto nos seus aspectos teóricos, quanto aplicados. Um tratamento profundo deste tema é o do livro de Lovász e Plummer (1986), inteiramente dedicado à teoria dos emparelhamentos. Os volumes enciclopédicos de Schrijver (2003), bem como os livros de Lawler (1976), Papadimitriou e Steiglitz (1982), Berge (1985) são textos de referência obrigatória. Entre os artigos tutoriais sobre emparelhamentos, recomendamos o de Gerards (1995). Um tutorial em língua portuguesa sobre algoritmos de emparelhamento é o de Figueiredo e Szwarcfiter (1999). A denominada *Condição de Hall*, descrita pelo Teorema 8.1, para emparelhamentos perfeitos em grafos bipartidos é devida a Hall (1935). Uma prova algorítmica deste resultado pode ser encontrada em Hall Jr. (1958). O Teorema 8.2, no qual se baseiam os algoritmos para determinação de emparelhamentos de cardinalidade máxima, é devido a Berge, C.Berge (1957). Veja também Berge, C.Berge (1962). O Teorema 8.3 é devido a König (1936). A complexidade de pior caso do Algoritmo 8.1, para a determinação do emparelhamento máximo de um grafo foi reduzida para  $O(n^{\frac{2}{3}})$  por Hopcroft e Karp (1973). O método Húngaro, bem como o seu respectivo algoritmo foram inicialmente apresentados por Kuhn (1955). O Algoritmo 8.2, para a determinação do emparelhamento ponderado máximo para grafos bipartidos é conhecido como Algoritmo Húngaro, e geralmente atribuído a Kuhn (1955) e Munkres (1957). Dentre os algoritmos existentes, figura o de Duan and Su (2012), dentre os mais eficientes. A teoria de floração, empregada para a determinação de emparelhamentos máximos em grafos gerais, é devida a Edmonds (1965). O Teorema 8.5 é parte desta teoria. O algoritmo para emparelhamento máximo em grafos gerais também se baseia no Teorema de Edmonds. A determinação de emparelhamentos máximos em grafos gerais permanece em evidência até os dias de hoje, continuamente. Gabow (1976) realizou uma implementação mais eficiente do algoritmo de Edmonds. A complexidade do algoritmo para grafos gerais foi reduzida para  $O(n^{\frac{5}{2}})$  por Even e Kariv (1975). Galil (1986) descreveu algoritmos eficientes para vários problemas de emparelhamentos. Micali e Vazirani (1980) descreveram o algoritmo de menor complexidade até hoje conhecido para emparelhamentos máximos em grafos gerais:  $O(m\sqrt{n})$ . A prova de correção deste último algoritmo, pela sua dificuldade, permanece como foco de atenção. Por exemplo, Vazirani (1994) publicou uma teoria de caminhos alternantes e florações, para a prova do algoritmo. O mesmo autor descreveu em Vazirani (2012) uma simplificação do algoritmo de Micali e Vazirani (1980), e sua prova. Além disso, em Vazirani (2014) o autor considera a prova do algoritmo original. Blum (2015) descreve sobre emparelhamentos máximos em grafos gerais, sem considerar explicitamente florações. Uma referência para o Exercício 8.19 é Petersen (1891). O método Húngaro, conjugado com o método primal-dual, de otimização combinatória, pode ser utilizado para resolver os problemas de emparelhamentos com arestas ponderadas, dos Exercícios 8.22 e 8.23. Além disso, o problema do Exercício 8.23 pode ser também resolvido através de caminhos aumentantes de peso máximo. Para esta última técnica, veja, por exemplo, o algoritmo descrito em Shier (2004).

Página deixada intencionalmente em branco

# PROBLEMAS NP-COMPLETOS

---

## 9.1 Introdução

Como motivação inicial a este capítulo, considere uma vez mais o problema de avaliar satisfatoriamente a eficiência de algoritmos. Em capítulos anteriores, foi ressaltada a conveniência de utilizar a complexidade como medida de eficiência. Contudo, uma vez de posse dessa complexidade de que forma reconhecer se o algoritmo correspondente é ou não eficiente?

O critério seguinte procura responder a esta nova questão. Um *algoritmo é eficiente precisamente quando a sua complexidade for um polinômio no tamanho de sua entrada*. Esta classificação certamente não é absoluta. De fato, pode ser até insatisfatória, por vezes. Contudo é aceitável para a grande maioria dos casos.

Seja agora a seguinte extensão do presente tópico. Considere a coleção de todos os algoritmos que resolvem um certo problema II. O interesse é conhecer se nessa coleção existe algum que seja eficiente, isto é, de complexidade polinomial. Se existir tal algoritmo, o problema II será denominado *tratável*, e *intratável* caso contrário. A ideia seria que um problema tratável pudesse sempre ser resolvido, para entradas e saídas de tamanho razoável, através de algum processo automático. Por exemplo, um computador. Enquanto isso, um algoritmo de complexidade não polinomial, de algum problema intratável, poderia em certos casos levar séculos para computar dados de entrada e saída de tamanhos relativamente reduzidos.

De acordo com a definição, um problema seria classificado como tratável, exibindo-se algum algoritmo de complexidade polinomial, que o resolvesse. Por outro lado, para verificar que é intratável, há necessidade de provar que todo possível algoritmo que o resolva não possui complexidade polinomial.

O presente capítulo apresenta uma introdução à teoria do NP-completo, na qual se consideram questões relativas à tratabilidade de problemas. De um modo geral, os problemas serão restritos a uma classe especial denominada problemas de decisão, apresentada na Seção 9.2. A classe P, compreendendo os problemas tratáveis, é examinada na Seção 9.3. Uma pequena lista de problemas aparentemente intratáveis é dada a seguir. A classe NP é o assunto da Seção 9.5, enquanto que a questão  $P = NP$  é apresentada em seguida. O complemento de um problema de decisão é o assunto da Seção 9.7. Em sequência, é introduzida a ideia de transformação polinomial entre problemas, a qual é utilizada para definir a classe NP-completo. Uma pequena lista desses é apresentada na Seção 9.9. As restrições e extensões de problemas são descritas na Seção 9.10, enquanto que o conceito de algoritmo pseudopolinomial encerra o capítulo.

## 9.2 Problemas de Decisão

De um modo geral, um *problema algorítmico* pode ser caracterizado por um conjunto de todos os possíveis dados do problema, denominado *conjunto de dados* e por uma questão solicitada, denominada *objetivo do problema*. *Resolver* o problema algorítmico consiste em desenvolver um algoritmo, cuja entrada são dados específicos reti-

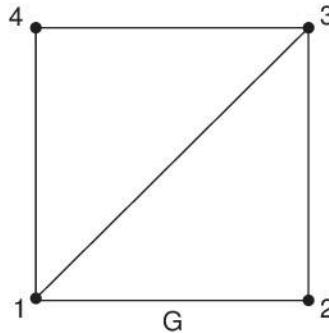


Figura 9.1: Instância do problema de clique

rados desse conjunto e cuja saída, denominada *solução*, responda ao objetivo do problema. Os dados específicos que constituem uma entrada formam uma *instância do problema*. Isto é, um problema possui tantas instâncias diferentes quantas são as variações possíveis de seus dados.

Assim, por exemplo, o problema algorítmico “elaborar um algoritmo para encontrar uma clique de tamanho  $\geq k$  num grafo  $G$  dado” pode ser colocado no seguinte formato (recorda-se que uma clique de  $G$ , de tamanho  $k$ , é um subgrafo completo de  $G$ , com  $k$  vértices):

DADOS: Um grafo  $G$  e um inteiro  $k > 0$

OBJETIVO: Encontrar em  $G$  uma clique de tamanho  $\geq k$ , se existir.

Assim, o conjunto de dados do problema algorítmico apresentado consiste no conjunto de todos os pares  $(G, k)$ , onde  $G$  é um grafo arbitrário e  $k$  um inteiro positivo arbitrário. Um par específico  $(G, k)$  constitui uma instância do problema. Um subgrafo completo de  $G$  com  $k$  ou mais vértices, se existir, é uma solução do problema.

Supõe-se que cada instância do problema seja apresentada ao algoritmo, segundo uma codificação conveniente. O comprimento total dessa codificação constitui o *tamanho* da entrada do algoritmo. Naturalmente, este parâmetro é importante, pois é em relação a ele que são tomadas as medidas de complexidade. Por esse motivo são inaceitáveis as codificações de instâncias que sejam desnecessariamente longas. De um modo geral, ela se torna desnecessariamente longa quando (i) contém partes irrelevantes ao problema ou (ii) um certo inteiro  $p$  da instância está codificado no sistema unário (isto é, no sistema de numeração da base 1, no qual cada inteiro  $p$  é codificado por  $p$  1's consecutivos). A condição (ii) exprime que são aceitáveis números na base 2, por exemplo. Naturalmente, à proporção que a base cresce, o tamanho da codificação decresce. Contudo, a relação entre os tamanhos das codificações de um mesmo número nas bases  $b_1$  e  $b_2$ , respectivamente, pode ser expressa por um polinômio, quando  $b_1, b_2 \geq 2$ . Enquanto isso, a relação é exponencial quando as bases são 2 e 1, o que justifica a condição (ii). A justificativa para a (i) é óbvia.

Como exemplo, seja o problema anterior, de encontrar no grafo  $G$ , uma clique de tamanho  $k > 0$ . Uma possível instância do problema é o grafo da Figura 9.1, e o número 3. Seja  $G$  fornecido através de seu conjunto de arestas  $\{(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)\}$ . Uma codificação aceitável para a entrada seria, por exemplo, /1,10 / 1,11 / 1,100 / 10,11 / 11,100 // 11 //. Nela, cada aresta  $(v, w)$  é codificada como  $/b_v, b_w/$ , onde  $b_v, b_w$  são, respectivamente, as representações binárias dos rótulos dos vértices  $v$  e  $w$ . O inteiro  $k$  aparece como  $//b_k//$ , sendo  $b_k$  a representação binária de  $k$ . O tamanho dessa instância é 35.

Há certas classes gerais de problemas algorítmicos. Por exemplo, existem os *Problemas de Decisão*, os de *Localização* e os de *Otimização*. Num problema de decisão, o objetivo consiste em decidir a resposta SIM ou NÃO a uma questão. Num problema de localização, o objetivo é localizar uma certa estrutura  $S$  que satisfaça um conjunto de propriedades dadas. Se as propriedades a que  $S$  deve satisfazer envolverem critérios  $C$  de otimização, então o problema torna-se de otimização. Observe que é possível formular um problema de decisão cujo objetivo é indagar se existe ou não a mencionada estrutura  $S$ , satisfazendo às propriedades dadas. Isto é, existem triplas de problemas, um de decisão, outro de localização e outro de otimização que podem ser *associados*, conforme indica a Figura 9.2.

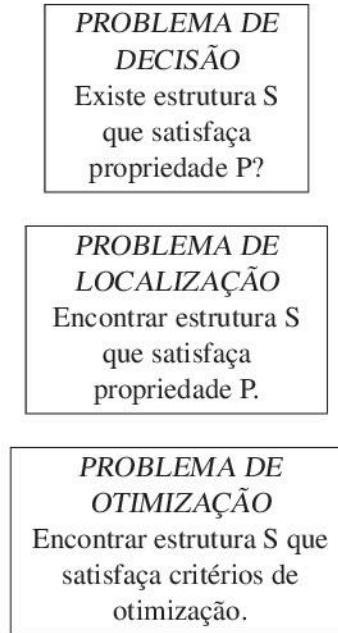


Figura 9.2: Problemas de decisão, localização e otimização associados

Como exemplo, considere o problema do *caixeiro viajante*. Seja  $G$  um grafo completo, tal que cada aresta  $e$  possui um peso  $c(e) \geq 0$ . Um *percurso de caixeiro viajante* é simplesmente um ciclo hamiltoniano de  $G$ . O *peso* de um percurso é a soma dos pesos das arestas que o formam. Um *percurso de caixeiro viajante ótimo* é aquele cujo peso é mínimo. Por exemplo, no grafo completo da Figura 9.3, os valores dos pesos estão indicados junto às arestas. Um percurso de caixeiro viajante é, por exemplo,  $a, b, c, d, a$ , cujo peso é 16, enquanto que um ótimo é  $a, b, d, c, a$  de peso 11.

Os seguintes problemas associados podem ser formulados:

#### PROBLEMA DE DECISÃO

DADOS: Um grafo  $G$  e um inteiro  $k > 0$

OBJETIVO: Verificar se  $G$  possui um percurso de caixeiro viajante de peso  $\leq k$ .

#### PROBLEMA DE LOCALIZAÇÃO

DADOS: Um grafo  $G$  e um inteiro  $k > 0$

OBJETIVO: Localizar, em  $G$ , um percurso de caixeiro viajante, de peso  $\leq k$ .

#### PROBLEMA DE OTIMIZAÇÃO

DADOS: Um grafo  $G$

OBJETIVO: Localizar, em  $G$ , um percurso de caixeiro viajante ótimo.

Os três problemas de caixeiro viajante, acima, estão obviamente relacionados. Suponha que o *Problema de Otimização* respetivo seja resolvido e denomine por  $Q$  o percurso ótimo encontrado. Então  $Q$  pode ser utilizado para resolver o *Problema de Localização* associado, da seguinte maneira: Seja  $c(Q)$  o peso do percurso  $Q$ . Note que  $c(Q)$  pode ser obtido facilmente (somando os pesos das arestas) de  $Q$ . Então se  $c(Q) \leq k$ ,  $Q$  é também uma solução para o *Problema de Localização*. Caso contrário,  $c(Q) > k$  e não existe em  $G$  percurso de caixeiro viajante de peso  $\leq k$ . Obviamente, isto resolve também o *Problema de Decisão* associado. Então, para o caso de caixeiro viajante, o *Problema de Decisão* é de dificuldade não maior do que o de *Localização*, e este de dificuldade não maior do que o de *Otimização*. Aliás, é natural que assim o seja. Contudo, é bastante menos intuitivo que,

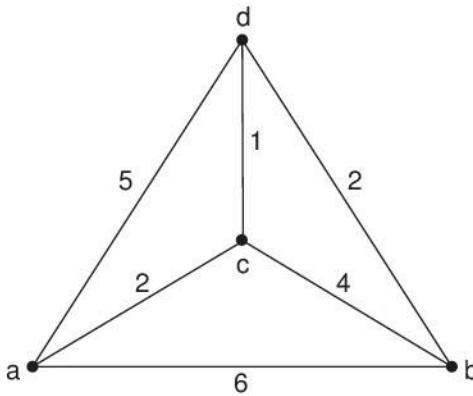


Figura 9.3: Problema do caixeteiro viajante

também em diversos casos, os problemas de otimização e localização apresente, ambos, dificuldades não maior do que o da decisão associado.

Os problemas em discussão neste capítulo serão todos de decisão. Uma justificativa para esta escolha é que, em geral, o problema de decisão é o mais simples dentre os três associados. Por isso, alguma prova de sua possível intratabilidade pode ser estendida aos outros casos.

A notação  $\Pi(D, Q)$  será utilizada para representar um problema de decisão  $\Pi$ , onde  $D$  representa o conjunto de dados e  $Q$  a questão (decisão) correspondente. Quando conveniente, utiliza-se  $\Pi(I)$  para denotar o problema  $\Pi(D, Q)$  aplicado à instância  $I$  de  $\Pi$ .

### 9.3 A Classe P

Conforme mencionado na Seção 9.1, um algoritmo eficiente é aquele cuja complexidade é uma função polinomial nos tamanhos dos dados de entrada. Observe que para um problema de decisão, o tamanho da saída é constante, o que possibilita ignorá-lo. Por exemplo, se  $n$  for o tamanho da entrada, algoritmos cujas complexidades sejam  $O(1)$ ,  $O(n)$ ,  $O(n^2 \log n)$  ou  $O(n^{10})$ , seriam todos classificados como eficientes. Por outro lado, complexidades como, por exemplo,  $O(2^n)$  ou  $O(n!)$  corresponderiam a algoritmos não eficientes.

Observe que o critério apresentado de avaliação de eficiência, certamente, não é absoluto. Por exemplo, considere dois algoritmos  $A$  e  $B$ , de complexidades  $O(n^{10})$  e  $O(2^n)$ , respectivamente, para um mesmo problema  $\Pi$ . O algoritmo  $A$  seria eficiente e  $B$  não eficiente. Suponha que  $A$  e  $B$  utilizem exatamente  $n^{10}$  e  $2^n$  passos, respectivamente, e ainda que ambos sejam implementados em um computador que efetua um passo em cada milissegundo. Para uma instância de  $\Pi$  em que  $n = 2$ , o algoritmo “eficiente” levaria  $2^{10} = 1024$  milissegundos, enquanto o “não eficiente” terminaria em  $2^2 = 4$  milissegundos. Contudo, é fácil verificar que quando  $n$  cresce, o algoritmo  $A$  melhora a sua *performance* em relação a  $B$  e a partir de certo ponto se torna, obviamente, mais eficiente.

O exemplo anterior, com  $n = 2$ , contudo, constitui exceção. Para a grande maioria dos casos práticos, os algoritmos de complexidade polinomial são, de um modo geral, eficientes. Enquanto que os de complexidade não polinomial não o são.

Para maior simplicidade de expressão um algoritmo será denominado *polinomial (exponencial)* quando sua complexidade for uma função polinomial (exponencial) nos tamanhos dos dados de entrada.

A adoção do critério anterior de classificação de algoritmos quanto a sua eficiência foi também motivada por outros fatores. Por exemplo, as expressões de complexidade dos algoritmos polinomiais são frequentemente polinômios de baixo grau. Isto é, são mais raras complexidades como  $O(n^{10})$  ou  $O(n^{12})$ . São comuns outras como  $O(n)$ ,  $O(n \log n)$  ou  $O(n^2)$ . Um outro argumento diz respeito ao modelo de computação correspondente à medida de complexidade. Como foi observado no Capítulo 1, a ideia de *passo* de um algoritmo é fundamental para a conceituação de sua complexidade. E esta ideia está relacionada ao *modelo de computação* utilizado. Existem alguns modelos que podem ser considerados como abstrações dos computadores reais (como o RAM, do Capítulo

1). Entre esses, em geral, é preservado o caráter polinomial de algoritmos. Isto é, um algoritmo polinomial, segundo um certo modelo, permaneceria polinomial quando traduzido para um outro. Assim sendo, o conceito de eficiência seria independente do modelo de computação adotado.

Com a motivação apresentada, define-se a classe P de problemas de decisão como sendo aquela que comprehende os problemas que admitem algoritmo polinomial. Por exemplo, existe um algoritmo (Seção 4.4) o qual verifica se um grafo é ou não biconexo, em complexidade linear no tamanho do grafo. Logo, o problema *Biconectividade* pertence à classe P.

Observe que se os algoritmos conhecidos para um certo problema  $\Pi$  forem todos exponenciais, não necessariamente  $\Pi \notin P$ . Se de fato  $\Pi \notin P$  então deve existir alguma *prova* de que todo possível algoritmo para resolver  $\Pi$  não é polinomial. Por exemplo, os algoritmos conhecidos até agora para o problema *Caixeiro Viajante*, são todos exponenciais. Contudo, não é conhecida prova de que seja impossível a formulação de algoritmo polinomial para o problema. Isto é, se desconhece se *Caixeiro Viajante* pertence ou não a P. Pode-se observar das seções seguintes que esta incerteza quanto à pertinência a P é compartilhada por um grande número de problemas.

## 9.4 Alguns Problemas Aparentemente Difíceis

Nesta seção apresenta-se uma lista de dez problemas. O primeiro deles é um problema de lógica, envolvendo expressões booleanas. Os demais são problemas conhecidos em grafos. Eles possuem em comum o fato de que são exponenciais todos os algoritmos desenvolvidos, até o momento, para resolver qualquer um deles. E isto, apesar do enorme esforço despendido por muitos, na tentativa de encontrar um algoritmo polinomial que resolvesse algum problema da lista. Por outro lado, também se desconhece até o momento qualquer prova de que tal algoritmo polinomial, de fato, não exista. Recorde-se, da seção anterior, que o problema *Caixeiro Viajante* possui esta mesma característica. Na realidade, como será observado mais adiante, há uma quantidade muito grande de outros problemas que compartilham, com os da lista a seguir, essa aparente intratabilidade.

### PROBLEMA 1: SATISFATIBILIDADE

DADOS: Uma expressão booleana E na FNC

DECISÃO: E é satisfatível?

Para enunciar o primeiro problema da lista, considere um conjunto de variáveis booleanas  $x_1, x_2, x_3, \dots$ , e denote por  $\bar{x}_1, \bar{x}_2, \bar{x}_3, \dots$  seus complementos. Isto é, cada variável  $x_i$  assume um valor *verdadeiro* (V) ou falso (F) e  $x_i$  é verdadeira se e somente se  $\bar{x}_i$  for falso. De um modo geral, as símbolos  $x_1, \bar{x}_1, x_2, \bar{x}_2, x_3, \bar{x}_3, \dots$  são denominados *literais*. Denote por  $\wedge$  e  $\vee$ , respectivamente as operações binárias usuais de conjunção (e) e disjunção (ou). A tabela da Figura 9.4 contém a definição dessas operações. Uma *cláusula* é uma disjunção de literais. Assim,  $\bar{x}_2 \vee x_3 \vee \bar{x}_4 \vee x_1$  é uma cláusula. Uma *expressão booleana* é uma expressão cujos operandos são literais e cujos operadores são conjunções ou disjunções. Uma expressão booleana é dita na *forma normal conjuntiva* (FNC) quando for uma conjunção de cláusulas. Assim, por exemplo:

$$(x_2 \vee \bar{x}_1) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_2) \wedge (x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_2)$$

é uma expressão booleana na FNC. Se atribuirmos um valor, verdadeiro ou falso, a cada variável de uma expressão booleana E, pode-se computar o *valor* (V ou F) de E, calculando-se os resultados das operações (conjunções e disjunções) indicadas na expressão. Assim, por exemplo, para a atribuição  $x_1 = F, x_2 = V, x_3 = V$ , a expressão anterior na FNC assume o valor V. Sabe-se que toda expressão booleana pode ser colocada na FNC, mediante a aplicação de transformações que preservam o seu valor. Uma expressão booleana é dita *satisfatível* se existe uma atribuição de valores, verdadeiro ou falso, às variáveis de tal modo que o valor da expressão seja verdadeiro. A expressão do exemplo apresentado é, pois, satisfatível. A expressão  $(x_1 \wedge \bar{x}_1)$  é obviamente não satisfatível, pois qualquer que seja o valor de  $x_1$ , a cláusula  $(x_1 \wedge \bar{x}_1)$  assume o valor falso. O *problema de satisfatibilidade* consiste em, dada uma expressão booleana na FNC, verificar se ela é satisfatível.

$x_1$	$x_2$	$x_1 \wedge x_2$	$x_1 \vee x_2$
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Figura 9.4: Conjunção e disjunção

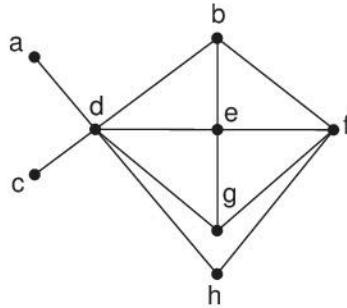


Figura 9.5: Exemplo para os problemas 2, 3 e 4

**PROBLEMA 2: CONJUNTO INDEPENDENTE DE VÉRTICES****DADOS:** Um grafo  $G$  e um inteiro  $k > 0$ **DECISÃO:**  $G$  possui um conjunto independente de vértices de tamanho  $\geq k$ ?

Dado um grafo  $G(V, E)$  recorda-se que um conjunto, independente de vértices é um subconjunto  $V' \subseteq V$  tal que todo par de vértices de  $V'$  não é adjacente. Isto é, se  $v, w \in V'$  então  $(v, w) \notin E$ . Por exemplo, no grafo da Figura 9.5,  $\{a, c, b, g, h\}$  é um tal conjunto, de cardinalidade 5. De fato, o maior desses no grafo da figura. O problema consiste em, dado  $G$ , verificar se o mesmo possui um conjunto independente de vértices de cardinalidade pelo menos igual a um valor dado.

**PROBLEMA 3: CLIQUE****DADOS:** Um grafo  $G$  e um inteiro  $k > 0$ **DECISÃO:**  $G$  possui uma clique de tamanho  $\geq k$ ?

Dado um grafo  $G(V, E)$  recorde-se que uma *clique* é um subconjunto  $V' \subseteq V$  tal que todo par de vértices de  $V'$  é adjacente. Isto é, se  $v, w \in V'$  então  $(v, w) \in E$ . Tal subconjunto induz, pois, um subgrafo completo. No grafo da Figura 9.5,  $\{d, b, e\}$  é uma clique de cardinalidade 3. De fato, a maior desse grafo. O problema em questão consiste em dado  $G$ , verificar se o mesmo possui uma clique de cardinalidade pelo menos igual a um valor dado.

**PROBLEMA 4: COBERTURA POR VÉRTICES****DADOS:** Um grafo  $G$  e um inteiro  $k > 0$ **DECISÃO:**  $G$  possui uma cobertura por vértices de tamanho  $\leq k$ ?

Para um grafo  $G(V, E)$ , um subconjunto  $V' \subseteq V$  é chamado *cobertura por vértices* quando toda aresta de  $G$  possuir (pelo menos) um de seus extremos em  $V'$ . Isto é, se  $(v, w) \in E$  então  $v$  ou  $w \in V'$ . Por exemplo,  $V' = \{d, f, e\}$  é uma cobertura por vértices de tamanho 3, do grafo da Figura 9.5. E, de fato, a cobertura de tamanho mínimo. O problema consiste em verificar se um grafo dado possui uma cobertura por vértices de tamanho no máximo igual a um valor dado.

**PROBLEMA 5: CICLO HAMILTONIANO DIRECIONADO**

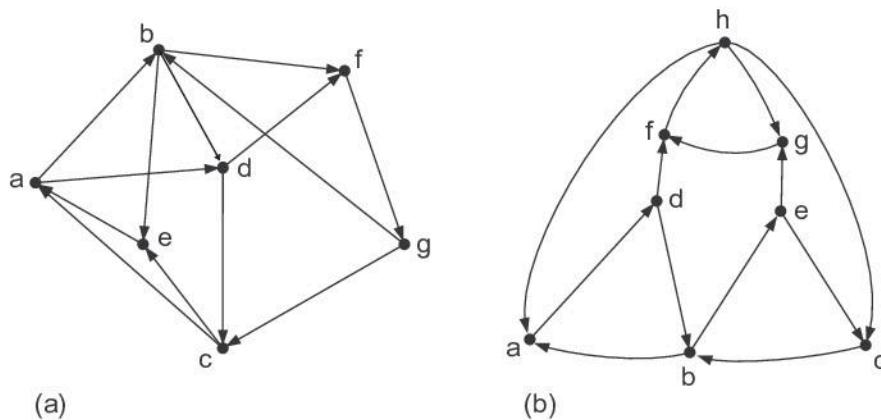


Figura 9.6: Exemplos para os problemas 5, 7, 8

DADOS: Digrafo  $D$ DECISÃO:  $D$  possui um ciclo hamiltoniano?

Recorde-se que um ciclo de um dado digrafo  $D$  é dito *hamiltoniano* quando ele contém exatamente uma vez cada vértice de  $D$ . O ciclo  $a, b, d, f, g, c, e, a$  do digrafo da Figura 9.6(a) é obviamente hamiltoniano, enquanto que o digrafo da Figura 9.6(b) não possui tal ciclo. O presente problema consiste em reconhecer digrafos hamiltonianos.

## PROBLEMA 6: CICLO HAMILTONIANO NÃO DIRECIONADO

DADOS: Grafo  $G$ DECISÃO:  $G$  possui um ciclo hamiltoniano?

Este problema é análogo ao anterior, exceto que o grafo não é direcionado.

## PROBLEMA 7: CONJUNTO DE ARESTAS DE REALIMENTAÇÃO

DADOS: Digrafo  $D$  e inteiro  $k > 0$ DECISÃO:  $D$  possui um conjunto de arestas de realimentação de tamanho  $\leq k$ ?

Para um digrafo  $D(V, E)$ , um *conjunto de arestas de realimentação* é um subconjunto  $E' \subseteq E$  tal que cada ciclo (direcionado) de  $D$  possui (pelo menos) uma aresta de  $E'$ . Ou seja,  $D(V, E - E')$  é acíclico. Na Figura 9.6(b), pode-se verificar que  $E' = \{(a, d), (f, h), (b, e)\}$  é um conjunto de arestas de realimentação, pois todo ciclo do digrafo exemplo contém pelo menos uma dessas três arestas. O presente problema consiste em verificar se um dado digrafo possui um conjunto de arestas de realimentação de tamanho no máximo igual a um valor dado.

## PROBLEMA 8: CONJUNTO DE VÉRTICES DE REALIMENTAÇÃO

DADOS: Digrafo  $D$ , número  $k > 0$ DECISÃO:  $D$  possui um conjunto de vértices de realimentação de tamanho  $\leq k$ ?

Este problema é análogo ao anterior, exceto que as arestas do conjunto  $E'$  são substituídas por vértices. Assim sendo, para um digrafo  $D(V, E)$  um *conjunto de vértices de realimentação* é um subconjunto  $V' \subseteq V$  tal que todo ciclo de  $D$  possui um vértice em  $V'$ . Ou seja, retirando-se os vértices  $V'$  de  $D$ , este se torna acíclico. No digrafo da Figura 9.6(b),  $V' = \{b, h\}$  é um conjunto de vértices de realimentação, de tamanho 2.

## PROBLEMA 9: COLORAÇÃO

DADOS: Um grafo  $G$  e um inteiro  $k > 0$

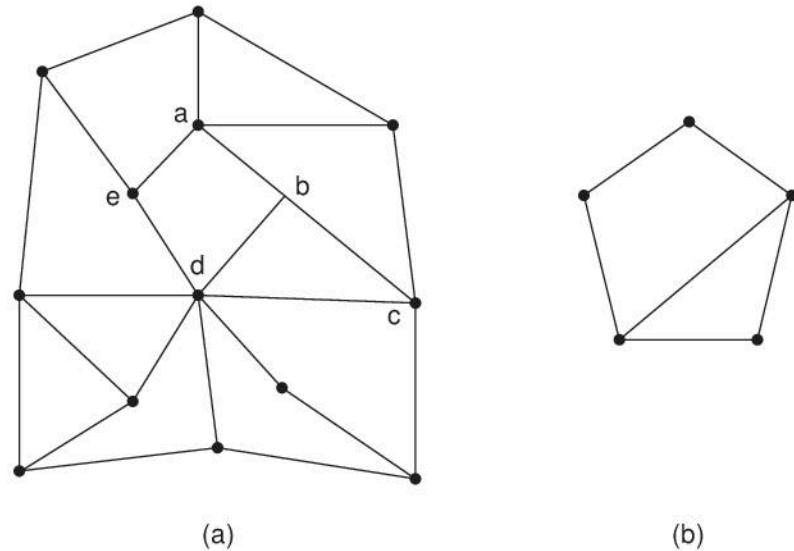


Figura 9.7: Exemplo para o problema 10

**DECISÃO:**  $G$  possui uma coloração com um número  $\leq k$  de cores?

Recorde-se que uma coloração de um grafo  $G(V, E)$  é uma atribuição de cores aos vértices de  $G$ , de tal modo que vértices adjacentes possuam cores diferentes. O problema em questão consiste em, dado um grafo  $G$ , verificar se o mesmo admite uma coloração que utiliza no máximo um número prefixado  $k$  de cores.

## PROBLEMA 10: ISOMORFISMO DE SUBGRAFOS

DADOS: Grafos  $G_1, G_2$

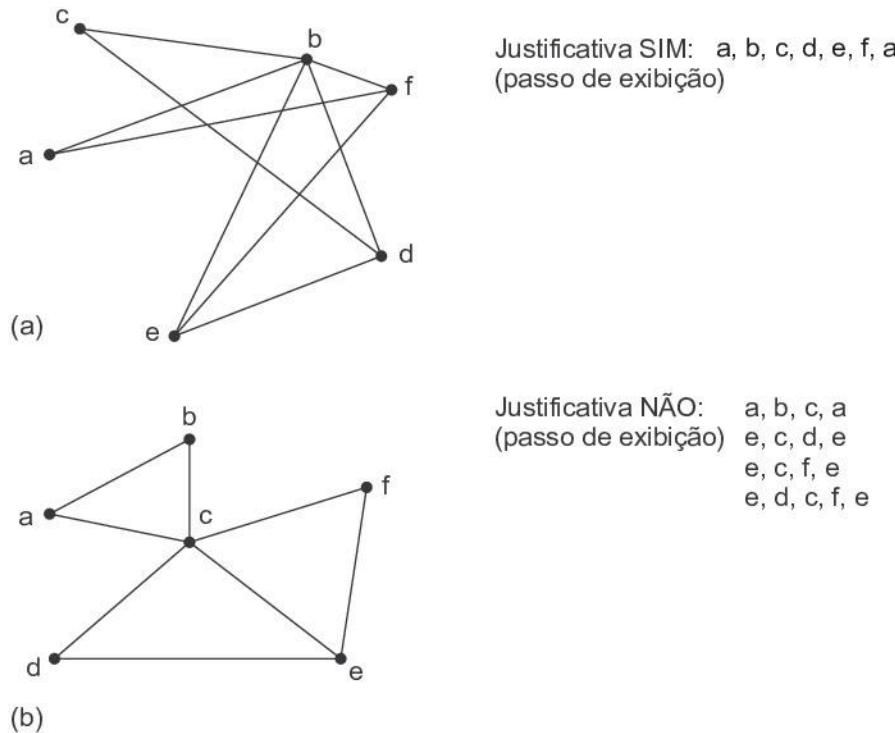
**DECISÃO:**  $G_1$  contém um subgrafo isomorfo a  $G_2$ ?

Considerando agora dois grafos  $G_1(V_1, E_1)$  e  $G_2(V_2, E_2)$ . Um subgrafo  $S(V'_1, E'_1)$  de  $G_1$  é dito isomorfo a  $G_2$  quando existe uma função unívoca  $f : V'_1 \rightarrow V_2$  que preserva adjacência. Isto é,  $(v, w) \in E'_1$  se e somente se  $(f(v), f(w)) \in E_2$ . Por exemplo o subgrafo induzido pelos vértices  $\{a, b, c, d, e\}$  do grafo da Figura 9.7(a) é isomorfo ao grafo da Figura 9.7(b). O problema presente consiste em, dados os grafos  $G_1$  e  $G_2$ , verificar se  $G_1$  contém um subgrafo isomorfo a  $G_2$ .

## 9.5 A Classe NP

Considere um problema de decisão  $\Pi$ . Se  $\Pi$  for solúvel através da aplicação de algum processo, então necessariamente existe uma *justificativa* para a solução de  $\Pi$ . Essa justificativa pode ser representada por um determinado conjunto de argumentos, os quais, quando interpretados convenientemente, podem atestar a veracidade da resposta SIM ou NÃO dada ao problema. Por exemplo, a solução do problema de verificar se um dado grafo é ou não biconexo pode ser obtida através do algoritmo da Seção 4.4. A justificativa para a solução encontrada é a própria correcção do algoritmo.

Considerando o problema *Ciclo Hamiltoniano* da seção anterior, onde o objetivo é decidir se um grafo  $G$  possui ou não ciclo hamiltoniano. Uma justificativa para a resposta SIM pode ser obtida, por exemplo, exibindo-se um ciclo  $C$  de  $G$  e reconhecendo-se que  $C$  é de fato hamiltoniano (ou seja, de que  $C$  é um ciclo e que contém todos os vértices de  $G$ , exatamente uma vez cada). Uma justificativa para a resposta NÃO pode ser apresentada, por exemplo, listando-se todos os ciclos simples de  $G$  e verificando-se que nenhum deles é hamiltoniano. Por exemplo, a resposta certa ao problema *Ciclo Hamiltoniano* para o grafo da Figura 9.8(a) é SIM. Como justificativa pode ser exibido o ciclo  $a, b, c, d, e, f, a$ . Verifica-se que o mesmo é hamiltoniano. Por outro lado, a resposta para

Figura 9.8: Justificativa para resposta ao problema *Ciclo Hamiltoniano*

o grafo da Figura 9.8(b) é NÃO. Como justificativa pode ser apresentada a seguinte lista de ciclos:  $\{a, b, c, a; e, c, d, e; e, c, f, e; e, d, c, f, e\}$ . Verifica-se que esta lista contém todos os ciclos simples do grafo e que nenhum deles é hamiltoniano.

Como ilustração adicional, considere o problema *Clique*, onde o objetivo é decidir se um dado grafo  $G$  contém uma clique de tamanho  $\geq k$ , onde  $k > 0$  é um inteiro dado. Uma justificativa para uma resposta SIM pode ser obtida exibindo-se uma clique  $P$  de tamanho  $\geq k$ . Efetua-se então uma verificação para reconhecer (i) que  $P$  de fato é uma clique (ou seja, todo par de vértices de  $P$  é adjacente em  $G$ ) e (ii) que  $|P| \geq k$ . Uma justificativa para uma resposta NÃO pode consistir em uma lista de todas as cliques de  $G$ . Efetua-se então uma verificação para confirmar que a lista de fato está completa e que o tamanho de cada clique é  $< k$ .

Observe que o processo apresentado de justificar respostas a problemas de decisão se compõe de duas fases distintas:

#### FASE 1: EXIBIÇÃO

Consiste em exibir a justificativa.

#### FASE 2: RECONHECIMENTO

Consiste em verificar que a justificativa apresentada (no passo de exibição) é, de fato, satisfatória.

Seja o problema *Ciclo Hamiltoniano* para o grafo da Figura 9.8(a), cuja solução é SIM. Na justificativa, o passo de exibição consiste na sequência  $C$  de vértices  $a, b, c, d, e, f, a$ . O passo de reconhecimento consiste em verificar se  $C$  é de fato um ciclo hamiltoniano. Como visto, o processo de reconhecimento é simples. Basta atestar: (i)  $C$  é um ciclo, e (ii)  $C$  contém cada vértice de  $G$  exatamente uma vez cada. O algoritmo correspondente ao processo é facilmente implementável em tempo polinomial no tamanho do grafo.

Retorne agora ao problema *Ciclo Hamiltoniano* para o grafo da Figura 9.8(b), cuja solução é NÃO. O passo de exibição da justificativa é o conjunto das 4 sequências:  $\{a, b, c, a; e, c, d, e; e, c, f, e; e, d, c, f, e\}$  conforme indicado. O passo de reconhecimento consiste em comprovar que (i) cada sequência de vértices é um ciclo não hamiltoniano e (ii) todo ciclo simples de  $G$  está presente na sequência exibida. O algoritmo de reconhecimento

não é tão simples quanto aquele da justificativa SIM. A operação (i) deve ser realizada para *cada* ciclo exibido (ao contrário do caso anterior que requeria verificações sobre um único). Para comprovar (ii) uma ideia seria enumerar *todos* os ciclos do grafo  $G$ .

Como o número total de ciclos pode ser exponencial com o tamanho de  $G$ , esse algoritmo é de natureza exponencial. Até o momento, é desconhecido se existe algum outro processo para justificar a resposta NÃO, do problema *Ciclo Hamiltoniano*, tal que o passo de reconhecimento corresponda a um algoritmo polinomial.

Define-se então a classe NP como sendo aquela que comprehende todos os problemas de decisão II, tais que existe uma justificativa à resposta SIM para II, cujo passo de reconhecimento pode ser realizado por um algoritmo polinomial no tamanho da entrada de II.

Ressalte-se na definição da classe NP, que não se exige uma solução polinomial para II. Além disso, para existir algoritmo de reconhecimento polinomial é necessário (mas não suficiente) que o *tamanho da justificativa*, dada pelo passo de exibição, seja polinomial no tamanho da entrada do problema. Por exemplo, a justificativa NÃO apresentada ao problema *Ciclo Hamiltoniano*, cuja entrada é o grafo da Figura 9.8(b), teve como passo de exibição uma lista de todos os seus ciclos. Conforme foi mencionado, o número de ciclos de um grafo  $G$  pode ser exponencial no tamanho de  $G$ . Portanto, essa justificativa é longa demais, o que implica que qualquer algoritmo de reconhecimento é exponencial no tamanho da entrada (apesar de existir algoritmo que seja polinomial no tamanho da justificativa). Ainda em relação à definição de NP, observe que nada se exige da justificativa NÃO. De fato, há problemas de NP que admitem algoritmos polinomiais para suas justificativas NÃO. Como há também aqueles para os quais tais algoritmos não são conhecidos.

Os exemplos de problemas pertencentes à classe NP são inúmeros. A justificativa SIM dada a *Ciclo Hamiltoniano* possui como passo de reconhecimento um algoritmo polinomial no tamanho da entrada do grafo. Logo, *Ciclo Hamiltoniano* pertence a NP.

Para verificar se um problema II pertence ou não a NP procede-se da seguinte maneira:

- (1) Define-se uma justificativa  $J$  conveniente para a resposta SIM ao problema;
- (2) Elabora-se um algoritmo para reconhecer se  $J$  está correta. A entrada desse algoritmo consiste em  $J$  e na entrada de II.

Se o algoritmo resultante do passo (2) for polinomial no tamanho da entrada de II, então II pertence a NP. O caso contrário, naturalmente, não implica necessariamente a não pertinência a NP.

Como ilustração do processo, mostra-se a seguir que os problemas *Satisfatibilidade*, *Conjunto Independente de Vértices* e *Cobertura por Vértices* pertencem a NP. Aliás, todos os demais problemas apresentados na seção anterior também pertencem.

**PROBLEMA: SATISFATIBILIDADE**

**DADOS:** Uma expressão  $E$  na FNC

**DECISÃO:**  $E$  é satisfatível?

**JUSTIFICATIVA SIM:**

- (1) **EXIBIÇÃO:** A expressão booleana  $E$  e uma atribuição para cada variável de  $E$ .
- (2) **ALGORITMO DE RECONHECIMENTO:** Substitui-se em  $E$  cada variável pelo seu valor atribuído (V ou F). É imediato concluir que a justificativa está correta se e só se cada cláusula de  $E$  possuir pelo menos uma variável com atribuição V.

**CONCLUSÃO:** O algoritmo de (2) é polinomial no tamanho dos dados de SATISFATIBILIDADE. Logo, este problema pertence a NP.

**PROBLEMA: CONJUNTO INDEPENDENTE DE VÉRTICES**

**DADOS:** Um grafo  $G(V, E)$  e um inteiro  $k > 0$

**DECISÃO:**  $G$  possui um conjunto independente de vértices, de tamanho  $\geq k$ ?

**JUSTIFICATIVA SIM:**

- (1) **EXIBIÇÃO:** O grafo  $G$  e um subconjunto de vértices  $V' \subseteq V$ .
- (2) **ALGORITMO DE RECONHECIMENTO:** Examina-se cada lista de adjacência  $A(v')$ ,  $v' \in V'$ , para verificar se todo  $w \in A(v')$  é tal que  $w \notin V'$ . Seja agora  $k' = |V'|$ . É imediato concluir que a justificativa está correta se e só se essas verificações forem todas satisfeitas, e além disso,  $k' \geq k$ .

**CONCLUSÃO:** O algoritmo (2) é polinomial no tamanho dos dados de CONJUNTO INDEPENDENTE DE VÉRTICES. Logo, pertence a NP.

**PROBLEMA: COBERTURA POR VÉRTICES**

**DADOS:** Um grafo  $G(V, E)$  e um inteiro  $k > 0$

**DECISÃO:**  $G$  possui uma cobertura por vértices de tamanho  $\leq k$ ?

**JUSTIFICATIVA: SIM**

- (1) **EXIBIÇÃO:** O grafo  $G$  e um subconjunto de vértices  $V' \subseteq V$ .
- (2) **ALGORITMO DE RECONHECIMENTO:** Examina-se cada aresta  $(v, w) \in E$  com o intuito de verificar se  $v$  ou  $w \in V'$ . Seja agora  $k' = |V'|$ . É imediato concluir que a justificativa está correta se e só se as verificações forem todas satisfeitas, e além disso,  $k' \leq k$ .

**CONCLUSÃO:** O algoritmo (2) é polinomial no tamanho dos dados COBERTURA POR VÉRTICES. Logo, pertence a NP.

Com essa mesma ideia, pode-se provar que todos os problemas apresentados na seção anterior pertencem a NP. E além daqueles, muitos outros. Contudo, existem também diversos problemas para os quais se desconhece sua pertinência ou não a essa classe. Um exemplo é a *Clique Máxima* a seguir.

**PROBLEMA: CLIQUE MÁXIMA**

**DADOS:** Um grafo  $G(V, E)$  e um inteiro  $k > 0$

**DECISÃO:** A clique de tamanho máximo de  $G$  possui  $k$  vértices?

Uma maneira de justificar uma resposta SIM ao problema apresentado pode ser assim descrita. No passo de exibição apresenta-se um conjunto  $S$  contendo todas as cliques maximais de  $G$ . No passo de reconhecimento, comprova-se que  $S$  contém de fato exatamente cada clique maximal de  $G$ . Seja  $k'$  o tamanho da maior clique de  $S$ . Obviamente, a justificativa está correta se e somente se  $k = k'$ . Como o tamanho de  $S$  pode ser exponencial no de  $G$ , este algoritmo é também exponencial. A conclusão natural é que a justificativa apresentada não permite afirmar que *Clique Máxima* pertence a NP. Isto não exclui a possibilidade de existir alguma outra que assim o permita. Contudo, tal justificativa não foi encontrada até o momento, mas também não foi provado que ela não possa existir. Consequentemente, não é conhecido se *Clique Máxima* pertence ou não a NP. Todas as evidências, no entanto, conduzem à conjectura de que *Clique Máxima*  $\notin$  NP.

Finalmente, embora não considerados neste texto, menciona-se que existem problemas para os quais são conhecidas provas de não pertinência a NP.

## 9.6 A Questão P = NP

Nessa seção discutem-se relações entre as classes P e NP. O lema seguinte é imediato.

**Lema 9.1** $P \subseteq NP$ .

**Prova** Seja  $\Pi \in P$  um problema de decisão. Então existe um algoritmo  $\alpha$  que apresenta a solução de  $\Pi$ , em tempo polinomial no tamanho de sua entrada. Em particular,  $\alpha$  pode ser utilizado como algoritmo de reconhecimento para uma justificativa à resposta SIM de  $\Pi$ . Logo,  $\Pi \in NP$ . ■

A pergunta natural seguinte seria  $P \neq NP$ ? Ou seja, existe algum problema na classe NP que seja intratável? Ou, caso contrário, todo problema em NP admite necessariamente algoritmo polinomial? Isto é, a exigência de que uma justificativa SIM pode ser reconhecida em tempo polinomial é suficiente para garantir a existência de um algoritmo polinomial?

Até o momento não se conhece a resposta a essa pergunta. Todas as evidências apontam na direção  $P \neq NP$ . Um argumento em favor dessa conjectura é que a classe NP incorpora um grande número de problemas, para os quais inúmeros pesquisadores já desenvolveram esforços para elaborar algoritmos eficientes. Apesar deste fato não foi possível a formulação de algoritmos polinomiais para qualquer deles. Os problemas desse conjunto pertenceriam então a  $NP - P$ . Na Seção 9.8, serão desenvolvidos argumentos adicionais que reforçam a ideia de que  $P \neq NP$ .

Uma outra questão, dessa vez simples de ser respondida, é a seguinte: Deseja-se saber *quão complexo* pode ser um problema da classe NP. Ou seja, admitindo que  $P \neq NP$ , seria possível ao menos resolver em tempo exponencial todo problema da classe NP? O Lema 9.2 responde afirmativamente a esta pergunta.

**Lema 9.2**

Seja  $\Pi \in NP$  um problema de decisão. Então uma resposta SIM ou NÃO para  $\Pi$  pode ser obtida em tempo exponencial com o tamanho de sua entrada.

**Prova** Se  $\Pi \in NP$ , então existe um algoritmo  $\alpha$  que reconhece se é verdadeira uma justificativa  $J$  para a resposta SIM a  $\Pi$ , tal que  $\alpha$  possui complexidade  $c_\alpha$ , polinomial no tamanho da entrada de  $\Pi$ . Seja  $k$  o tamanho de  $J$ , o qual é necessariamente também polinomial na entrada de  $\Pi$ . Ora,  $J$  é escrita através de uma sequência apropriada que contém letras, números ou símbolos especiais. Existe um conjunto  $A$  de cardinalidade fixa o qual contém todos os elementos que podem compor  $J$ . A ideia é gerar todas as sequências possíveis de comprimento  $k$  que podem ser formadas com elementos de  $A$ . Para cada uma dessas sequências aplica-se o algoritmo  $\alpha$ . Então  $\Pi$  possui resposta SIM se e somente se pelo menos uma das saídas de  $\alpha$  for SIM.  $\alpha$  é aplicado um total de  $|A|^k$  vezes. Portanto, foi descrito um algoritmo para  $\Pi$  de complexidade  $O(|A|^k c_\alpha)$ . Isto prova o lema. ■

## 9.7 Complementos de Problemas

Conforme mencionado, a definição da classe NP exige que a justificativa SIM deva ser reconhecida em tempo polinomial, enquanto nada se exige da justificativa NÃO. Este fato sugere a possibilidade de se inverter os papéis desempenhados pelas mesmas, o que conduz a uma nova classe de problemas. Assim sendo, define-se a classe Co-NP como sendo aquela que comprehende todos os problemas de decisão  $\Pi$ , tais que existe uma justificativa à resposta NÃO, cujo passo de reconhecimento corresponde a um algoritmo polinomial no tamanho da entrada de  $\Pi$ .

Por analogia, define-se o *complemento*  $\bar{\Pi}$  de um problema de decisão  $\Pi$  como sendo o problema de decisão cujo objetivo é o complemento da decisão de  $\Pi$ . Ou seja,  $\Pi$  e  $\bar{\Pi}$  possuem SIM e NÃO trocados. Isto é, a resposta ao problema  $\Pi$  é SIM se e somente se a resposta a  $\bar{\Pi}$  for NÃO.

Uma consequência direta dessas definições é que a classe Co-NP comprehende exatamente os complementos dos problemas da classe NP. Ou seja,  $\Pi \in NP$  se e somente se  $\bar{\Pi} \in Co-NP$ .

Pode-se formular para a classe Co-NP um lema semelhante ao 9.1. Ou seja, é imediato verificar que  $P \subseteq Co-NP$ . Assim sendo, se um problema de decisão  $\Pi$  admitir solução polinomial, então obviamente  $\Pi \in P$  e  $\Pi \in NP \cap Co-NP$ . Por outro lado, existem problemas  $\Pi \in NP$  para os quais é desconhecido se  $\bar{\Pi}$  também pertence ou não a essa classe. Por analogia, é possível construir exemplos de problemas da classe Co-NP, mas cuja pertinência ou não

a Co-NP de seus complementos seja desconhecida. Há também casos para os quais se desconhece a pertinência ou não a NP, tanto de  $\Pi$  quanto do complemento  $\bar{\Pi}$ .

Como exemplo, considere o problema CLIQUE, anteriormente mencionado. Seu complemento é o problema CLIQUE seguinte:

**PROBLEMA: CLIQUE**

**DADOS:** Um grafo  $G(V, E)$  e um inteiro  $k > 0$

**DECISÃO:**  $G$  não possui clique de tamanho  $\geq k$ ?

Já foi observado que  $\text{CLIQUE} \in \text{NP}$ . Para que CLIQUE também pertença a essa classe seria necessária a existência de um algoritmo que reconhecesse em tempo polinomial com o tamanho de  $G$ , uma conveniente justificativa SIM à pergunta formulada na decisão anterior. Contudo, conforme já foi observado, não é conhecido algoritmo para tal reconhecimento, que seja polinomial no tamanho do grafo. Mas tampouco se conhece qualquer prova de sua não existência. Consequentemente, não é sabido se CLIQUE  $\in \text{NP}$ .

Um exemplo para o qual é desconhecida a pertinência a NP tanto de  $\Pi$ , quanto de  $\bar{\Pi}$  é a *Clique Máxima*, formulada na Seção 9.5. Exemplos de problemas tais que ambos  $\Pi$  e  $\bar{\Pi}$  pertencem a NP são inúmeros. Como foi mencionado,  $P \subseteq \text{NP}$  e  $P \subseteq \text{Co-NP}$ . Logo, qualquer problema  $\Pi \in P$  satisfaz  $\Pi \in \text{NP}$  e  $\bar{\Pi} \in \text{NP}$ .

Um caso interessante é o problema NÚMEROS COMPOSTOS, definido a seguir.

**PROBLEMA: NÚMEROS COMPOSTOS**

**DADOS:** Um número inteiro  $k > 0$

**DECISÃO:** Existem inteiros  $p, q > 1$  tais que  $k = pq$ ?

É imediato verificar que  $\text{NÚMEROS COMPOSTOS} \in \text{NP}$ . Para tanto, basta observar que o passo de exibição de uma justificativa SIM consiste em listar os inteiros  $p$  e  $q$ . O passo de reconhecimento corresponde a efetuar o produto  $p \cdot q$  e compará-lo com  $k$ . Por outro lado, o complemento de NÚMEROS COMPOSTOS é o seguinte:

**PROBLEMA: NÚMEROS COMPOSTOS (NÚMEROS PRIMOS)**

**DADOS:** Um número  $k > 0$

**DECISÃO:**  $k$  é primo?

Mais recentemente, foi descoberto um algoritmo de tempo polinomial para decidir o problema anterior, isto é, verificar se um número é primo. Isto permite concluir que  $\text{NÚMEROS COMPOSTOS} \in P$ .

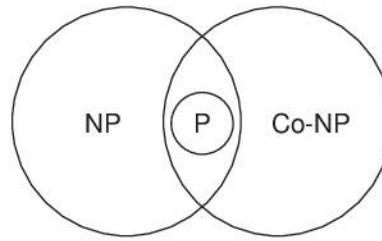
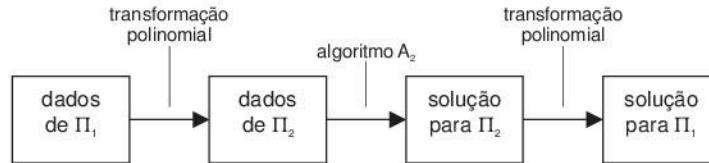
As seguintes questões não foram resolvidas até o momento:

- (i)  $\text{NP} = \text{Co-NP}$ ?
- (ii)  $P = \text{NP} \cap \text{Co-NP}$ ?

Observe que as expressões (i) e (ii) acima não são independentes da questão  $P = NP$ ? De fato, se  $P = NP$ , então necessariamente  $\text{NP} = \text{Co-NP}$  e portanto ambas (i) e (ii) possuem resposta SIM. Contudo, outras combinações são também possíveis. Conjectura-se que de fato  $\text{NP} \neq \text{Co-NP}$  e  $P \neq \text{NP} \cap \text{Co-NP}$ . A Figura 9.9 ilustra as conjecturas.

## 9.8 Transformações Polinomiais

Sejam  $\Pi_1(D_1, Q_1)$  e  $\Pi_2(D_2, Q_2)$  problemas de decisão. Suponha que seja conhecido um algoritmo  $A_2$  para resolver  $\Pi_2$ . Se for possível transformar o problema  $\Pi_1$  em  $\Pi_2$  e sendo conhecido um processo de transformar a solução de  $\Pi_2$  numa solução de  $\Pi_1$ , então o algoritmo  $A_2$  pode ser utilizado para resolver o problema  $\Pi_1$ . Observe que como o objetivo consiste em resolver  $\Pi_1$ , através de  $A_2$ , o que se supõe dado é uma instância de  $\Pi_1$  para a qual se deseja conhecer a resposta. A partir da instância  $I_1 \in D_1$ , elabora-se a instância  $I_2 \in D_2$ . Aplica-se então o algoritmo  $A_2$  para resolver  $\Pi_2$ . O retorno ao problema  $\Pi_1$  é realizado através da transformação da

Figura 9.9: Conjecturas  $P \neq NP$ ,  $NP \neq Co\text{-}NP$ ,  $P \neq NP \cap Co\text{-}NP$ Figura 9.10: Transformação polinomial do problema  $\Pi_1$  em  $\Pi_2$ 

solução de  $\Pi_2$  para a de  $\Pi_1$ . Se a transformação de  $\Pi_1$  em  $\Pi_2$ , bem como a da solução de  $\Pi_2$  na de  $\Pi_1$ , puder ser realizada em tempo polinomial, então diz-se que existe uma *transformação polinomial* de  $\Pi_1$  em  $\Pi_2$ , e que  $\Pi_1$  é *polinomialmente transformável* em  $\Pi_2$ .

Formalmente, uma *transformação polinomial* de um problema de decisão  $\Pi_1(D_1, Q_1)$  no problema de decisão  $\Pi_2(D_2, Q_2)$  é uma função  $f : D_1 \rightarrow D_2$  tal que as seguintes condições são satisfeitas:

- (i)  $f$  pode ser computada em tempo polinomial, e
- (ii) para toda instância  $I \in D_1$  do problema  $\Pi_1$  tem-se:  $\Pi_1(I)$  possui resposta SIM se e somente se  $\Pi_2(f(I))$  também o possuir.

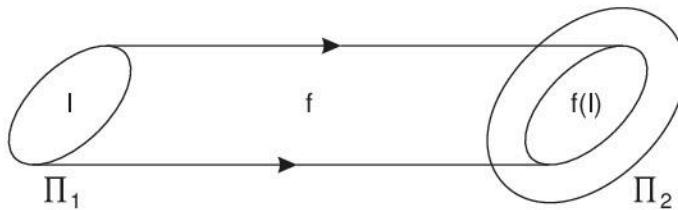
A Figura 9.10 descreve o processo. Observe que as transformações de maior interesse são precisamente as polinomiais. Isto ocorre porque essas últimas preservam a natureza (polinomial ou não) do algoritmo  $A_2$  para  $\Pi_2$ , quando utilizado para resolver  $\Pi_1$ . Assim sendo, se  $A_2$  for polinomial, e existir uma transformação polinomial de  $\Pi_1$  em  $\Pi_2$  então  $\Pi_1$  também pode ser resolvido em tempo polinomial. Denota-se  $\Pi_1 \propto \Pi_2$  para indicar que  $\Pi_1$  pode ser transformado polinomialmente em  $\Pi_2$ . Observe que a relação  $\propto$  é transitiva (isto é,  $\Pi_1 \propto \Pi_2$  e  $\Pi_2 \propto \Pi_3 \Rightarrow \Pi_1 \propto \Pi_3$ ).

Para ilustrar um processo simples de transformação polinomial, considere como  $\Pi_1$  e  $\Pi_2$ , respectivamente, os problemas CLIQUE e CONJUNTO INDEPENDENTE DE VÉRTICES, formulados na Seção 9.4. A instância  $I$  de CLIQUE consiste em um grafo  $G(V, E)$  e um inteiro  $k > 0$ . Para obter a instância  $f(I)$  de CONJUNTO INDEPENDENTE, considera-se o grafo complemento  $\overline{G}$  de  $G$  e o mesmo inteiro  $k$ . É então evidente que  $f$  é uma transformação polinomial porque:

- (i)  $\overline{G}$  pode ser obtido a partir de  $G$ , em tempo polinomial, e
- (ii)  $G$  possui uma clique de tamanho  $\geq k$  se e somente se  $\overline{G}$  possui um conjunto independente de vértices de tamanho  $\geq k$ .

Portanto, se existir um algoritmo  $A_2$  que resolve CONJUNTO INDEPENDENTE em tempo polinomial, este algoritmo pode ser utilizado para resolver CLIQUE também em tempo polinomial. Então CLIQUE  $\propto$  CONJUNTO INDEPENDENTE.

Observe que quando  $\Pi_1 \propto \Pi_2$  a transformação  $f$  deve ser aplicada sobre uma instância  $I$  genérica de  $\Pi_1$ . A instância que se obtém de  $\Pi_2$  contudo é particular, pois é fruto da transformação  $f$  utilizada. Assim sendo, de certa forma,  $\Pi_2$  pode ser considerado como um problema de dificuldade maior ou igual a  $\Pi_1$ . Pois que, mediante  $f$ , um algoritmo que resolve a instância particular de  $\Pi_2$ , obtida através de  $f$ , resolve também a instância arbitrária de  $\Pi_1$  dada.

Figura 9.11: Equivalência entre  $\Pi_1(D_1, Q_1)$  e  $(f(D_1), Q_2)$ 

Considere agora dois problemas  $\Pi_1$  e  $\Pi_2$ , tais que  $\Pi_1 \propto \Pi_2$  e  $\Pi_2 \propto \Pi_1$ . Então  $\Pi_1$  e  $\Pi_2$  são denominados problemas equivalentes. Segue-se que, segundo a observação anterior, dois problemas equivalentes são de idêntica dificuldade (no que se refere à existência ou não de algoritmo polinomial para resolvê-los). Note que quando  $\Pi_1 \propto \Pi_2$ , a particularização de  $\Pi_2$  construída pela transformação  $f$  é equivalente ao problema geral  $\Pi_1$ . Isto é, representando por  $f(D_1)$  o conjunto imagem de  $f : D_1 \rightarrow D_2$ , o problema  $\Pi_1(D_1, Q_1)$  é equivalente à particularização  $(f(D_1), Q_2)$ , do problema  $\Pi_2(D_2, Q_2)$ , conforme ilustra a Figura 9.11.

Observe que é possível utilizar a relação  $\propto$  para dividir NP em classes de problemas equivalentes entre si. Obviamente, os problemas pertencentes a P formam uma dessas classes. E, nesse sentido, podem ser considerados como os de “menor dificuldade” em NP.

Em contrapartida, existe outra classe de problemas equivalentes entre si, que correspondem aos de “maior dificuldade” dentre todos em NP. Os problemas dessa nova classe são denominados *NP-completos*, cuja definição se segue. Um problema de decisão  $\Pi$  é denominado NP-completo quando as seguintes condições forem ambas satisfeitas:

- (i)  $\Pi \in \text{NP}$
- (ii) todo problema de decisão  $\Pi' \in \text{NP}$  satisfaz  $\Pi' \propto \Pi$ .

Observe que (ii) implica que todo problema da classe NP pode ser transformado polinomialmente no problema  $\Pi$  NP-completo. Isto é, se um problema NP-completo  $\Pi$  puder ser resolvido em tempo polinomial então *todo* problema de NP admite também algoritmo polinomial (e consequentemente, nesta hipótese,  $P = \text{NP}$ ). Vale, pois,  $\Pi \in \text{P}$  se e somente se  $P = \text{NP}$ . Isto justifica o fato de que a classe NP-completo corresponde aos problemas de maior dificuldade dentre os pertencentes a NP. Caso somente a condição (ii) da definição de NP-completo seja considerada (não importando se (i) é ou não satisfeita), o problema  $\Pi$  é denominado *NP-difícil*. Consequentemente, a “dificuldade” de um problema NP-difícil é não menor do que a de um NP-completo.

## 9.9 Alguns Problemas NP-Completos

Uma vez definida a classe NP-completo, o passo natural seguinte consiste em identificar problemas que pertençam a essa classe. Como a única informação disponível até o momento é a definição de NP-completo, o processo de identificação de tais problemas envolveria a aplicação dessa definição. Contudo, esbarra-se numa dificuldade. A condição (ii) anterior requer que seja provado que *todo* problema de NP pode ser polinomialmente transformado no problema candidato a pertencer à classe. Esta tarefa seria por demais árdua para ser aplicada a cada possível candidato a NP-completo. O lema seguinte vem contornar esta questão.

### Lema 9.3

Sejam  $\Pi_1, \Pi_2$  problemas de decisão  $\in \text{NP}$ . Se  $\Pi_1$  é NP-completo e  $\Pi_1 \propto \Pi_2$  então  $\Pi_2$  é também NP-completo.

**Prova** Como  $\Pi_2 \in \text{NP}$ , para mostrar que  $\Pi_2$  é NP-completo, basta provar que  $\Pi' \propto \Pi_2$ , para todo  $\Pi' \in \text{NP}$ . Como  $\Pi_1$  é NP-completo, então necessariamente  $\Pi' \propto \Pi_1$ . Como  $\Pi_1 \propto \Pi_2$ , por transitividade tem-se  $\Pi' \propto \Pi_2$ . ■

O Lema 9.3, apesar da simplicidade, é muito poderoso. Como consequência, para provar que um certo problema  $\Pi$  é NP-completo, ao invés de utilizar a mencionada condição (ii) da definição (a qual exige uma prova

de que todo problema  $\Pi' \in NP$  é polinomialmente transformável em  $\Pi$ ), basta demonstrar que *um* problema NP-completo é polinomialmente transformável em  $\Pi$ . Ou seja, é suficiente provar que

- (i)  $\Pi \in NP$ , e
- (ii) um problema NP-completo  $\Pi'$  é tal que  $\Pi' \propto \Pi$ .

Observe que se apenas (ii) for realizado, então  $\Pi$  é NP-difícil, mas não necessariamente NP-completo. Contudo, para que este esquema de prova possa ser utilizado, é necessário escolher algum problema  $\Pi'$  que seja NP-completo. Portanto, para se identificar o *primeiro* problema desta classe, o processo apresentado não se aplica. Essa primeira identificação é dada pelo Teorema 9.1.

### Teorema 9.1

O problema SATISFATIBILIDADE (Seção 9.4) é NP-completo.

A prova do teorema foge ao escopo deste texto. Ela consiste em uma transformação polinomial genérica, de um problema da classe NP em SATISFATIBILIDADE.

Uma vez identificado um primeiro membro da classe NP-completo, o reconhecimento de outros pode ser realizado da maneira mais simples, através da aplicação do processo anterior. Assim sendo, tem-se:

### Teorema 9.2

O problema CLIQUE é NP-completo.

**Prova** Os dados de CLIQUE são um grafo e um inteiro positivo. Seja  $C$  uma clique do grafo. Pode-se reconhecer se  $C$  é uma clique e computar o seu tamanho, em tempo polinomial no tamanho da entrada de CLIQUE. Logo,  $CLIQUE \in NP$ . É necessário agora verificar que algum problema NP-completo pode ser polinomialmente transformável em CLIQUE. Seja uma instância genérica de SATISFATIBILIDADE. Esta é constituída de uma expressão booleana  $E$  na FNC, compreendendo as cláusulas  $L_1, \dots, L_p$ . A questão de decidir se  $E$  é ou não satisfatível será transformada numa questão de decidir se um certo grafo  $G(V, E)$  possui ou não uma CLIQUE de tamanho  $\geq p$ . O grafo  $G$  é construído, a partir da expressão  $E$ , da seguinte maneira: Existe um vértice diferente  $v_i$  em  $G$ , para cada ocorrência  $x_i$  de literal em  $E$ . Existe uma aresta diferente  $(v_i, v_j)$  em  $G$ , para cada par de literais  $x_i, x_j$  de  $E$ , tais que o literal  $x_i$  não representa  $\bar{x}_j$ , e  $x_i, x_j$  ocorrem em cláusulas diferentes de  $E$ . Como decorrência, cada aresta  $(v_i, v_j)$  de  $G$  é tal que os literais  $x_i, x_j$ , correspondentes em  $E$ , estão em cláusulas diferentes e podem assumir o valor verdadeiro simultaneamente. Logo, uma clique de  $G$  com  $p$  vértices corresponde em  $E$ , a  $p$  literais, um em cada cláusula, os quais podem assumir o valor verdadeiro simultaneamente. E reciprocamente. Portanto, decidir se  $E$  é satisfatível é equivalente a decidir se  $G$  possui uma clique de tamanho  $\geq p$  (Figura 9.12). Para completar a prova, basta observar que a construção de  $G$  pode ser facilmente realizada em tempo polinomial com o tamanho de  $E$ . ■

### Teorema 9.3

O problema CONJUNTO INDEPENDENTE é NP-completo.

**Prova** Na Seção 9.5 foi provado que  $CONJUNTO\ INDEPENDENTE \in NP$ . Na Seção 9.8 foi verificado que  $CLIQUE \propto CONJUNTO\ INDEPENDENTE$ . Sabe-se pelo Teorema 9.2 que CLIQUE é NP-completo. Logo, CONJUNTO INDEPENDENTE é NP-completo. ■

### Teorema 9.4

COBERTURA POR VÉRTICES é NP-completo.

**Prova** É imediato verificar que  $COBERTURA\ POR\ VÉRTICES \in NP$ . Considere uma instância genérica de CONJUNTO INDEPENDENTE, o qual é NP-completo (Teorema 9.3). Esta se compõe de um grafo  $G$  e de um inteiro  $k > 0$ . Seja transformar esta instância numa de COBERTURA POR VÉRTICES, cuja entrada consiste em

$$E = (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

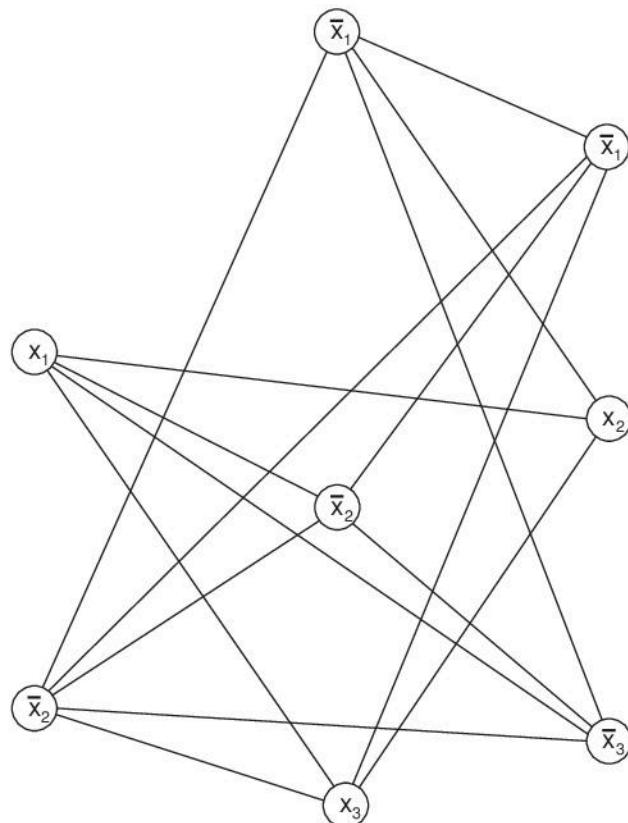


Figura 9.12: Transformação da prova do Teorema 9.2

um grafo  $G'$  e de um inteiro  $p > 0$ . Define-se então  $G' = G$  e  $p = |V| - k$  (note, que  $k < |V|$ , caso contrário COBERTURA POR VÉRTICES torna-se trivial). Se CONJUNTO INDEPENDENTE possui resposta SIM, existe um conjunto independente  $S$ , tal que  $|S| \geq k$ . Então o conjunto  $V - S$  é uma cobertura por vértices de  $G$ , de tamanho  $\leq p$ . Reciprocamente, se  $C$  é uma cobertura por vértices tal que  $|C| \leq p$ , então obviamente  $V - C$  é um conjunto independente de tamanho  $\geq k$ . Logo, CONJUNTO INDEPENDENTE  $\propto$  COBERTURA POR VÉRTICES. ■

Segundo uma sequência similar de passos, pode-se mostrar que todos os problemas enunciados na Seção 9.4 são NP-completos. E naturalmente a lista de tais problemas não se resume aos daquela seção. A literatura já aponta várias centenas desses. Eles podem ser encontrados em áreas relativamente diversas, como grafos, teoria dos números, lógica, construção de software, programação matemática e outras. Nesse conjunto, encontram-se alguns problemas para os quais já há dezenas de anos existem tentativas de desenvolvimento de algoritmos eficientes. O resultado infrutífero dessas tentativas, pelo menos até o momento, bem como o grande número de problemas da lista constituem indícios da provável correção da afirmativa  $P \neq NP$ .

## 9.10 Restrições e Extensões de Problemas

Sejam  $\Pi(D, Q)$  e  $\Pi'(D', Q')$  problemas de decisão. Diz-se que  $\Pi'$  é uma *restrição* de  $\Pi$  quando  $D' \subseteq D$  e  $Q' = Q$ . Analogamente, o problema  $\Pi$  é chamado de *extensão* de  $\Pi'$ . Isto é, a questão a ser respondida para ambos os problemas é a mesma. Mas os dados de  $\Pi'$  são retirados de um conjunto  $D'$  igual ou mais restrito do que o conjunto  $D$ , de onde os dados de  $\Pi$  são retirados.

Como exemplo, considere o problema a seguir.

**PROBLEMA: 3-SATISFATIBILIDADE**

**DADOS:** Uma expressão booleana  $E$  na FNC, tal que cada cláusula de  $E$  possui exatamente 3 literais

**DECISÃO:**  $E$  é satisfatível?

Por exemplo,  $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$  é uma instância do problema apresentado. Obviamente, as questões de decisão de SATISFATIBILIDADE e 3-SATISFATIBILIDADE são idênticas. Além disso, cada instância desse último é uma expressão booleana na FNC, cujas cláusulas possuem exatamente 3 literais cada. Ou seja, o conjunto de dados desse último é um subconjunto dos dados de SATISFATIBILIDADE (cujas expressões booleanas na FNC possuem cláusulas com número arbitrário de literais). Logo, 3-SATISFATIBILIDADE é uma restrição de SATISFATIBILIDADE.

Considere agora o problema CLIQUE. Uma maneira alternativa de formular CLIQUE consiste em considerar seus dados como um grafo  $G$  e uma clique  $C$ . A questão agora se constitui em indagar se  $G$  admite  $C$  como subgrafo. Obviamente, esse problema é de fato uma simples reformulação de CLIQUE, como definido, anteriormente, na Seção 9.2. Pois saber se um grafo  $G$  possui uma clique com pelo menos  $k$  vértices é exatamente o mesmo que saber se  $G$  possui um subgrafo isomorfo à clique  $C$ , possuindo  $|C| = k$  vértices. Assim sendo, CLIQUE é uma restrição de ISOMORFISMO DE SUBGRAFOS. Em ambos, as instâncias são dadas por dois grafos  $G$  e  $H$ , sendo  $G$  arbitrário. Para CLIQUE,  $H$  é uma clique, enquanto que para ISOMORFISMO DE SUBGRAFOS,  $H$  é também arbitrário. Logo, o conjunto de dados do primeiro é um subconjunto dos dados de ISOMORFISMO DE SUBGRAFOS. E além disso, a decisão “ $G$  possui um subgrafo isomorfo a  $H$ ?” é comum a ambos.

A aplicação principal desse conceito é devido ao lema seguinte:

### Lema 9.4

Sejam  $\Pi, \Pi'$  dois problemas de decisão tais que  $\Pi'$  é uma restrição de  $\Pi$ , e  $\Pi'$  é NP-completos. Então  $\Pi$  é NP-difícil.

**Prova** Basta mostrar que  $\Pi'$  é polinomialmente transformável em  $\Pi$ . Considere para  $f$ , a identidade que associa a cada instância  $I$  de  $\Pi'$ , uma imagem  $f(I) = I$  em  $\Pi$ . Então  $\Pi' \propto \Pi$ , o que prova o lema. ■

Naturalmente, se no Lema 9.4  $\Pi$  pertencer a NP então  $\Pi$  é NP-completo. Observe também que quando o problema de decisão  $\Pi'$  for uma restrição de  $\Pi$  então um algoritmo polinomial que resolva  $\Pi$ , resolverá também  $\Pi'$ . Por outro lado, o fato de  $\Pi$  ser NP-completo não implica, necessariamente, que  $\Pi'$  também o seja. O Lema 9.4 sugere um método alternativo para provar que um certo problema  $\Pi$  é NP-completo. Ou seja,  $\Pi$  será NP-completo quando:

- (i)  $\Pi$  pertencer à classe NP, e
- (ii) existir uma restrição  $\Pi'$  de  $\Pi$ , que seja NP-completo.

Observe que para provar (ii), a ideia consiste em a partir de  $\Pi$  procurar algum problema  $\Pi'$  NP-completo, e tal que  $\Pi'$  seja uma restrição de  $\Pi$ . Este procedimento contrasta com a metodologia de prova exposta na seção anterior, a qual tinha como ponto de partida a escolha de algum problema NP-completo, para tentar transformá-lo em  $\Pi$ . Muitas vezes, a alternativa anterior conduz a provas mais simples do que as obtidas com o procedimento da seção anterior.

Como exemplo, considere novamente os problemas CLIQUE e ISOMORFISMO DE SUBGRAFOS. Pelo Teorema 9.2, CLIQUE é NP-completo. Conforme exposto, sabe-se que CLIQUE é uma restrição de ISOMORFISMO DE SUBGRAFOS. Por outro lado, não é difícil concluir que ISOMORFISMO DE SUBGRAFOS pertence à classe NP. Isto prova o seguinte lema.

### **Teorema 9.5**

O problema ISOMORFISMO DE SUBGRAFOS é NP-completo.

Considere agora os problemas 3-SATISFATIBILIDADE e SATISFATIBILIDADE. Conforme mencionado, o primeiro desses é uma restrição do segundo. Sabe-se também que SATISFATIBILIDADE é NP-completo. Portanto, a aplicação do Lema 9.4 nada permite afirmar com relação a conhecer se 3-SATISFATIBILIDADE é ou não NP-completo. Assim sendo, para verificar que este problema é de fato NP-completo, utiliza-se a metodologia da seção anterior.

### **Teorema 9.6**

O problema 3-SATISFATIBILIDADE é NP-completo.

**Prova** É imediato que 3-SATISFATIBILIDADE pertence a NP. Porque SATISFATIBILIDADE, que é uma extensão desse problema, também o pertence. O passo seguinte é estabelecer uma transformação polinomial de algum problema NP-completo em 3-SATISFATIBILIDADE. Considere então uma instância genérica de SATISFATIBILIDADE, dada por uma expressão booleana  $E$ , na FNC. Substitui-se cada cláusula  $(g_1 \vee g_2 \vee \dots \vee g_k)$  de  $E$ , onde  $g_i$  são os literais e  $k \geq 4$ , pelo par de cláusulas  $(g_1 \vee g_2 \vee h) \wedge (g_3 \vee \dots \vee g_k \vee \bar{h})$ , onde  $h$  é uma nova variável. Repete-se o processo até que todas as cláusulas possuam não mais de 3 literais cada. Agora substitui-se cada cláusula  $(g_1 \vee g_2)$  com 2 literais, pelo par de cláusulas  $(g_1 \vee g_2 \vee h) \wedge (\bar{h} \vee \bar{h} \vee \bar{h})$ , onde  $h$  é uma nova variável. Procedimento correspondente é efetuado para cláusulas contendo um único literal. Desse modo, obtém-se uma expressão  $E'$  na FNC, em que cada cláusula contém exatamente 3 literais. É simples verificar que  $E'$  é satisfatível se e somente se  $E$  o for. Além disso, a construção de  $E'$ , a partir de  $E$ , pode ser realizada em tempo polinomial com o tamanho de  $E$ . Isto completa a prova. ■

Uma questão natural, nesse momento, seria considerar o problema 2-SATISFATIBILIDADE, que é uma restrição de SATISFATIBILIDADE, tal que cada cláusula possui exatamente 2 literais. Novamente, a aplicação do Lema 9.4 não traz informação para que se possa concluir se o mesmo é ou não NP-completo. Sabe-se, porém, que existe um algoritmo polinomial que o resolve. Portanto 2-SATISFATIBILIDADE pertence a P. Considere agora o problema  $k$ -SATISFATIBILIDADE,  $k > 3$ , que é a restrição de SATISFATIBILIDADE, em que cada cláusula possui exatamente  $k$  literais. Como 3-SATISFATIBILIDADE é NP-completo, a aplicação do Lema 9.4 permite concluir que  $k$ -SATISFATIBILIDADE também é NP-completo.

### **Teorema 9.7**

3-coloração é NP-completo.

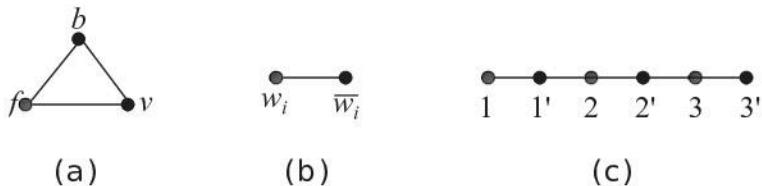


Figura 9.13: Vértices de  $G$

**Prova** É bastante simples verificar que 3-coloração  $\in NP$ . A transformação polinomial escolhida é a partir do problema 3-SATISFATIBILIDADE. Seja  $B = C_1 \cup \dots \cup C_n$  uma expressão booleana na forma normal conjuntiva, composta de  $n$  cláusulas  $C_j$ , cada qual composta exatamente por 3 literais. A partir da expressão booleana  $B$ , construiremos um grafo  $G$ . Este contém um triângulo especial, denominado base, cujos vértices são rotulados com  $v, b, f$  (Figura 9.13(a)). A ideia é que, na 3-coloração de  $G$ , a esses vértices sejam atribuídas as cores

- vértice v — cor verde  
vértice b — cor branca  
vértice f — cor fúcsia

A Figura 9.13(a) ilustra o triângulo base, (b) o artefato de variáveis, e (c) o artefato de cláusulas. Cada variável  $x_i$  de  $B$  corresponde, em  $G$ , a um par de vértices adjacentes  $w_i, \bar{w}_i$ , denominado artefato de variável (Figura 9.13(b)). Cada cláusula  $C_j$  corresponde, no grafo, a um caminho de 6 vértices,  $1, 1', 2, 2', 3, 3'$ , denominado artefato de cláusula (Figura 9.13(c)). Estes são os vértices de  $G$ .

Além das arestas descritas,  $G$  contém as seguintes arestas adicionais.

Os pares de vértices adjacentes  $w_i, \overline{w_i}$  de cada artefato de variável  $x_i$  são ligados ao vértice  $b$  do triângulo base (Figura 9.13(a)). Por outro lado, os vértices de cada artefato de literal  $C_j$  dão origem às seguintes arestas adicionais:

- Os vértices  $1, 1'$  são ligados ao vértice correspondente ao primeiro literal de  $C_j$ . Os vértices  $2, 2'$  são ligados ao vértice correspondente ao segundo literal de  $C_j$ . Os vértices  $3, 3'$  são ligados ao vértice correspondente ao terceiro literal de  $C_j$ . Os vértices  $1, 3'$  são ligados ao vértice  $v$  do triângulo base.

Ver a Figura 9.14(b).

A Figura 9.14(a) ilustra a ligação entre artefatos de variáveis e  $b$ , e a Figura 9.14(b) mostra as ligações entre os vértices do artefato do literal  $C_j = \{x_i, \overline{x_k}, x_p\}$ , e os vértices dos artefatos das variáveis  $x_i, x_k, x_p$ , bem como o vértice  $v$  do triângulo base.

A construção de  $G$  está completa.

Como exemplo, seja a expressão booleana  $B = C_1 \wedge C_2 \wedge C_3$ , onde  $C_1 = (x_1 \vee x_2 \vee \overline{x_3})$ ,  $C_2 = (\overline{x_1} \vee x_2 \vee \overline{x_3})$ ,  $C_3 = (x_1 \vee \overline{x_2} \vee x_3)$ .

O grafo  $G$  obtido através da construção apresentada está ilustrado na Figura 9.15, onde é considerado a expressão booleana  $B = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$ .

A próxima etapa da prova consiste em mostrar que  $B$  é satisfatível se e somente se  $G$  for 3-colorível.

Primeira parte: Suponha que  $G$  seja 3-colorível.

Seja uma 3-coloração de  $G$ , com as cores branca ( $b$ ), verde ( $v$ ) e fúcsia ( $f$ ). Os vértices  $w_i, \bar{w}_i$  de cada variável  $x_i$  não estão coloridos com a cor branca, pois ambos são adjacentes ao vértice  $b$  do triângulo base. Restam, portanto, as cores verde e fúcsia para  $w_i, \bar{w}_i$ . Observe que  $w_i, \bar{w}_i$  possuem cores distintas. Definimos uma atribuição para as variáveis  $x_i$  de  $\beta$ , que dependerá de cor  $w_i$  em  $G$ , do seguinte modo:

$$\begin{cases} \text{cor}(w_i) = \text{verde} \Rightarrow \text{Definir } x_i \text{ como verdadeiro em } B \\ \text{cor}(w_i) = \text{fúcsia} \Rightarrow \text{Definir } x_i \text{ como falso em } B \end{cases}$$

Vamos mostrar que a atribuição apresentada torna  $B$  satisfatível. Suponha que não. Então existe uma cáusula  $C_j$  em que as três variáveis são falsas. A situação é ilustrada na Figura 9.16(a). Observe que os vértices correspondentes ao artefato de cáusula  $C_j$  estão todos ligados ao vértice de cor fúcsia do triângulo base, e seus extremos

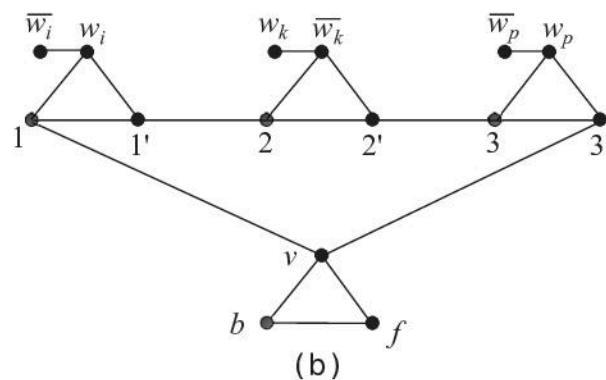
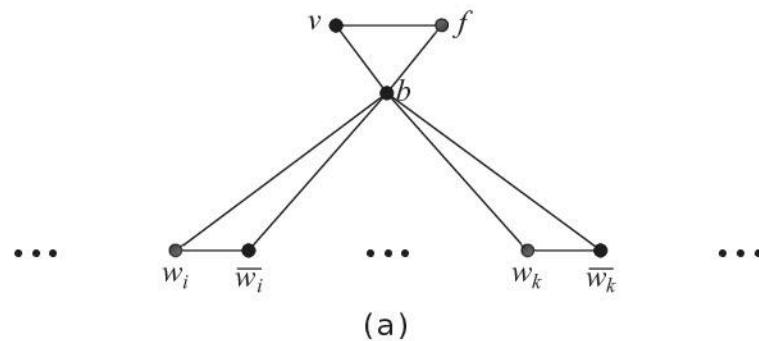
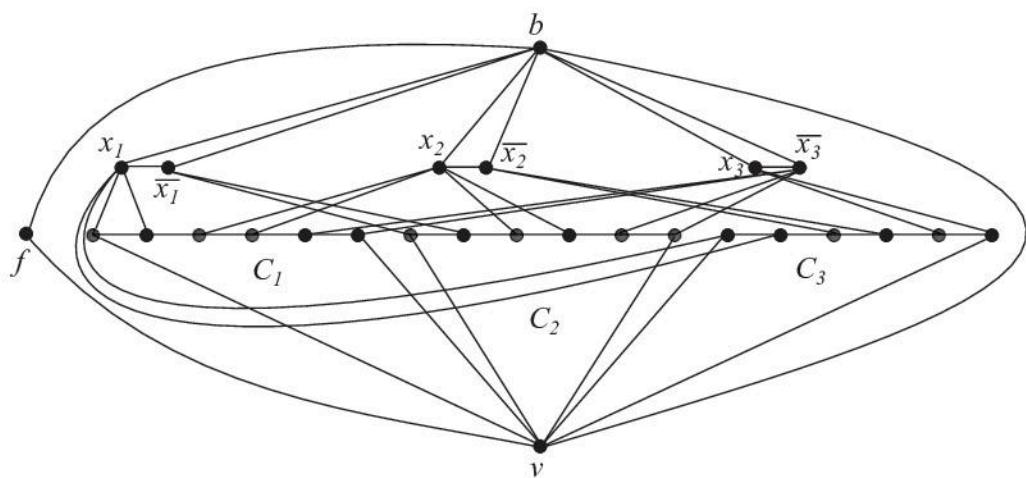
Figura 9.14: Arestas adicionais de  $G$ 

Figura 9.15: Grafo da prova do Teorema 9.7

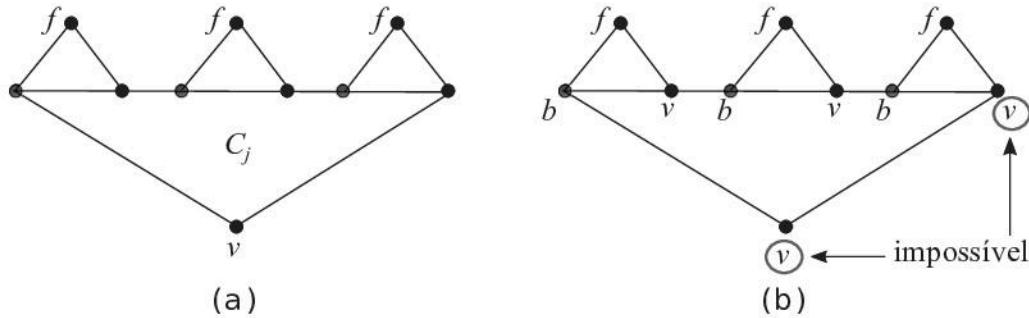


Figura 9.16: Contradição na prova do Lema 9.4.

estão ligados aos vértices de cor branca do triângulo base (Figura 9.16(a)). Isto implica uma coloração forçada dos vértices correspondentes ao artefato de cláusula  $C_j$ , quando percorridas da esquerda para a direita, conforme indica a Figura 9.16(b). Nesse caso, ao vértice da extrema direita do artefato da cláusula é atribuída a cor verde, que é idêntica à cor do vértice do triângulo base, ao qual ele é ligado. Isto contradiz a 3-coloração de  $G$ . Portanto,  $B$  é certamente satisfatível.

Segunda Parte: Suponha que  $B$  seja satisfatível.

Vamos mostrar que  $G$  é 3-colorível, apresentando uma 3-coloração de  $G$ . Sabemos que a cada variável  $x_i$  de  $B$  está associada um valor verdadeiro ou falso. Vamos definir a seguinte coloração da  $G$ .

Triângulo base: colorir o vértice  $v$  com a cor verde, o vértice  $b$  com branco e  $f$  com fúcsia.

Artefatos de cláusula: Resta, apenas, colorir os vértices  $1, 1', 2, 2', 3, 3'$  do artefato de cada cláusula  $C_j$ . Sabemos que os vértices de cada par  $1, 1'; 2, 2';$  e  $3, 3'$  estão ligados a um vértice comum, já colorido através dos artefatos de variáveis. Assim há formação de 3 triângulos, cujas bases ainda não foram coloridas, e cujos topo já foram, através da atribuição de cores já realizadas. Sabemos também que os vértices  $1, 3'$  estão ligados ao vértice do triângulo base, colorido com verde. A situação é ilustrada na Figura 9.17(a). De acordo com a Primeira Parte desta prova sabemos também que os tops desses três triângulos não estão coloridos simultaneamente, com a cor fúcsia. Além disso, pela construção de  $G$ , todos os tops estão ligados ao vértice branco do triângulo base, o que implica que nenhum dos tops é branco. Restam então 5 possibilidades para as cores dos tops. Para cada uma delas, mostramos uma coloração possível para os vértices  $1, 1', 2, 2', 3, 3'$ , como se segue:

Caso 1: Cores dos tops  $v, v, v$ . Ver Figura 9.17(b).

Caso 2: Cores dos tops  $v, v, f$ . Ver Figura 9.17(c).

Caso 3: Cores dos tops  $v, f, v$ . Ver Figura 9.17(d).

Caso 4: Cores dos tops  $v, f, f$ . Ver Figura 9.17(e).

Caso 5: Cores dos tops  $f, v, f$ . Ver Figura 9.17(f).

Com isso, obtemos uma 3-coloração própria para o grafo  $G$ , em qualquer caso, o que completa a prova do Teorema 9.7

## 9.11 Algoritmos Pseudopolinomiais

Na Seção 5.5, foi apresentado um algoritmo de programação dinâmica para o problema de o particionamento de árvores. O problema de decisão correspondente pode ser formulado como:

### PROBLEMA: PARTICIONAMENTO DE ÁRVORES

DADOS: Uma árvore  $T(V, E)$ , um peso  $d(e)$  para cada aresta  $e \in E$ , um peso inteiro  $z(v)$  para cada vértice  $v \in V$ , inteiros  $k$  e  $m > 0$ .

DECISÃO: Existe um particionamento da árvore  $T$  em subárvores disjuntas,  $T_1(V_1, E_1), \dots, T_t(V_t, E_t)$ , tal que:

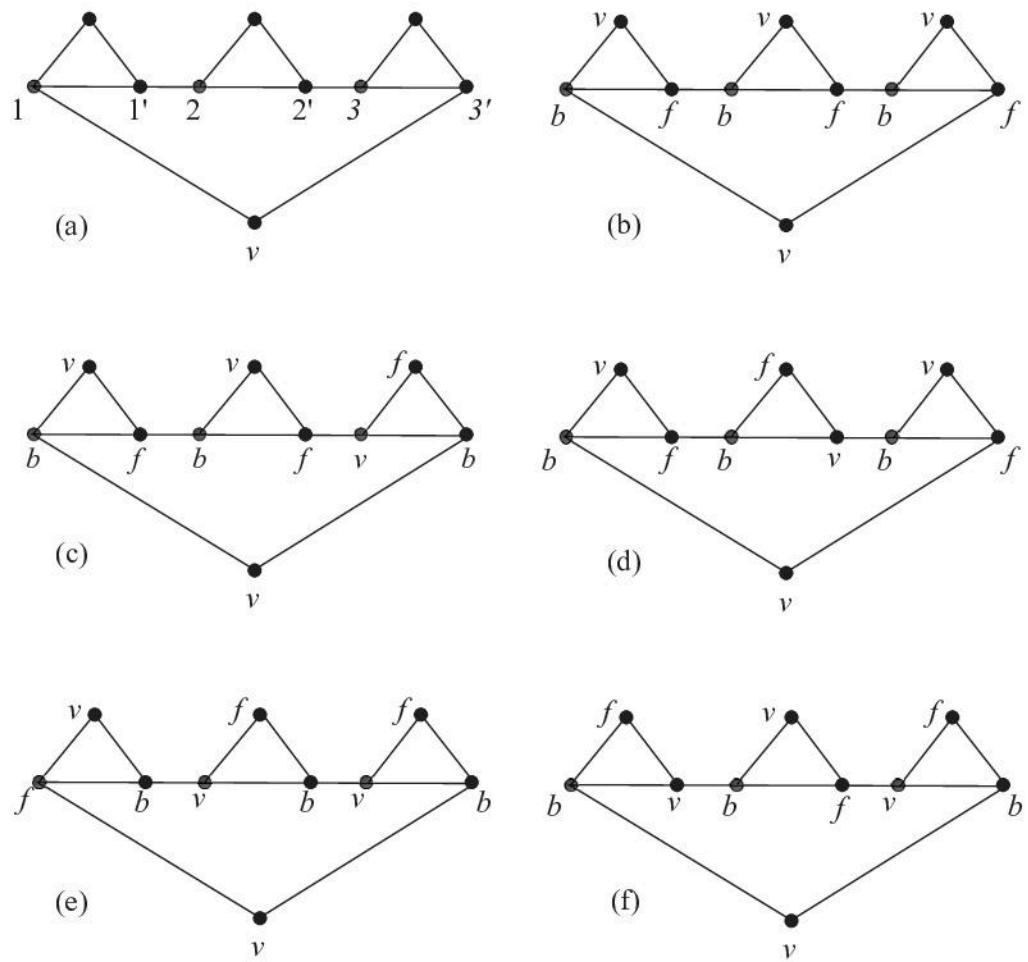


Figura 9.17: Atribuição de cores do Lema 9.7.

- (i)  $\sum_{v \in V_i} z(v) \leq k$ , para cada  $i = 1, \dots, t$ , e
- (ii)  $\sum_{e \notin E_i} d(e) \leq m$ , para  $i = 1, \dots, t$ ?

Em outras palavras, o problema **PARTICIONAMENTO DE ÁRVORES** consiste em indagar se é possível dividir uma árvore dada  $T$  em subárvore disjuntas  $T_1, \dots, T_t$ , de tal modo que (i) a soma dos pesos dos vértices em cada subárvore  $T_i$  não excede um valor  $k$  dado e (ii) a soma dos pesos das arestas não pertencentes a qualquer das subárvore não excede um valor  $m$  dado. Em relação a este problema pode-se provar o seguinte:

### Teorema 9.8

O problema **PARTICIONAMENTO DE ÁRVORES** é NP-completo.

A prova segue a estratégia geral apresentada na Seção 9.9, mas será omitida neste texto.

O problema de otimização associado ao **PARTICIONAMENTO DE ÁRVORES**, anteriormente formulado, foi considerado no final da Seção 5.5 como extensão do problema de particionamento lá considerado. A solução indicada por esta extensão particionava uma árvore em subárvore, de modo que (i) a soma dos pesos dos vértices em cada subárvore não excede o valor  $k$  dado e (ii) a soma dos pesos das arestas não pertencentes a qualquer subárvore é mínima. Portanto, aquele processo (Exercício 9.6) resolve também o problema de decisão **PARTICIONAMENTO DE ÁRVORES**. A complexidade do algoritmo citado é  $O(nk^2)$ , onde  $n = |V|$  e  $k$  é o inteiro dado. Mas, então, não seria este um algoritmo polinomial?

Para justificar o motivo pelo qual o mencionado processo *não* é polinomial, recorda-se que a complexidade de um algoritmo é calculada em função do tamanho da entrada. Isto é, um algoritmo polinomial possui como expressão de complexidade um polinômio no tamanho de sua entrada. Contudo, esse tamanho é naturalmente função da codificação adotada. Sabe-se que a representação binária é aceitável como critério de codificação. Além disso, uma codificação em uma base  $b > 2$  é obviamente também aceitável, pois não altera a natureza (polinomial ou exponencial) do algoritmo. Contudo, a codificação unária ( $b = 1$ ) não é aceita porque pode alterar a tal natureza. Na base 2, o número  $k$  seria representado por uma sequência de  $1 + \lfloor \log_2 k \rfloor$  dígitos. Consequentemente,  $O(nk^2)$  não é polinomial no tamanho da entrada de **PARTICIONAMENTO DE ÁRVORES**.

Observe que  $O(nk^2)$  seria de fato um algoritmo polinomial, caso a entrada de **PARTICIONAMENTO DE ÁRVORES** fosse escrita em representação unária. Por outro lado, há outros problemas NP-completos para os quais não podem existir algoritmos polinomiais (a menos que  $P = NP$ ), mesmo se a entrada for codificada em unário. Isto sugere as seguintes definições.

Seja  $\Pi$  um problema de decisão. Um algoritmo  $A$  que resolva  $\Pi$  é dito *pseudopolinomial* quando a complexidade de  $A$  for polinomial no tamanho da entrada de  $\Pi$ , supondo que esta seja codificada em unário. Por exemplo, o algoritmo mencionado para a extensão do problema de particionamento da Seção 5.5 é pseudopolinomial.

Um problema NP-completo que admite algoritmo pseudopolinomial para solucioná-lo é denominado *NP-completo fraco*. Assim sendo, **PARTICIONAMENTO DE ÁRVORES** é um exemplo de NP-completo fraco. Por outro lado, existem problemas NP-completos cuja possível existência de algoritmos pseudopolinomiais implicaria a igualdade  $P = NP$ . Esses últimos são chamados *NP-completos forte*.

Existem problemas para os quais o valor do maior dado de entrada a ser codificado é um polinômio na quantidade total de dados. Por exemplo, o problema **CICLO HAMILTONIANO** pertence a esta classe. Pois sua entrada consiste apenas em um grafo  $G$ . Para codificar  $G$  não há necessidade de utilizar valores numéricos maiores do que  $n = |V|$  ou  $m = |E|$ . Por outro lado, há necessidade de representar uma quantidade total de, pelo menos,  $\max(n, m)$  dados, implícita ou explicitamente. Nesse caso, a representação unária de cada elemento da entrada não pode alterar a natureza (polinomial ou exponencial) do algoritmo. Isto porque a quantidade total de dados a serem representados não é afetada pela codificação particular utilizada. Para esta classe de problemas, um algoritmo pseudopolinomial seria, de fato, um algoritmo polinomial. Portanto, todos os problemas NP-completos, com tal característica, são NP-completos forte.

Há outros problemas tais que, para uma instância arbitrária, o valor do maior dado *não* está limitado por um polinômio na quantidade total de dados. Esses últimos são denominados *problemas numéricos*. Por exemplo, **PARTICIONAMENTO DE ÁRVORES** pertence a esta classe. Isto porque o valor do parâmetro  $k$  pode ser arbitrariamente grande, impedindo no caso geral a possibilidade de expressá-lo através de um polinômio no número de

vértices da árvore. Assim sendo, os problemas numéricos são os únicos candidatos a admitir algoritmos pseudopolinomiais. E de fato, há exemplos dos dois casos possíveis. Isto é, problemas numéricos que são NP-completos fraco (como PARTICIONAMENTO DE ÁRVORES), bem como os NP-completos forte (como CAIXEIRO VIAJANTE).

Observe que um algoritmo pseudopolinomial para um problema II torna-se de fato polinomial para uma instância  $I$  particular quando o valor do maior dado de  $I$  for polinomial com a quantidade de dados. Assim, por exemplo, no problema PARTICIONAMENTO DE ÁRVORES, quando todos os pesos dos vértices forem unitários, o problema torna-se restrito ao caso  $k < n$  (pois se  $k \geq n$ , a solução é trivialmente SIM, consistindo em uma única partição com todos os vértices da árvore). Nessa hipótese, a complexidade  $O(nk^2)$  é de fato polinomial no tamanho da entrada. Ou seja, o algoritmo correspondente torna-se polinomial.

## 9.12 Complexidade de um Problema Numérico

Nesta seção, examinamos com detalhe um problema numérico, denominado SUBCONJUNTO SOMA. Será apresentada uma prova de sua NP-completude e, em seguida, um algoritmo pseudopolinomial para resolvê-lo. A conclusão é que se trata de um problema NP-completo fraco.

**PROBLEMA: SUBCONJUNTO SOMA**

**DADOS:** Um subconjunto finito  $S$ , de números naturais, com possíveis repetições, e um número natural  $t$ .

**DECISÃO:** Existe subconjunto  $S' \subseteq S$ , tal que  $\sum_{s \in S'} s = t$ ?

Obviamente, trata-se de um problema numérico, pois cada valor  $s \in S$ , não está limitado, necessariamente, por um polinômio em  $|S|$ .

### Teorema 9.9

SUBCONJUNTO SOMA é NP-completo.

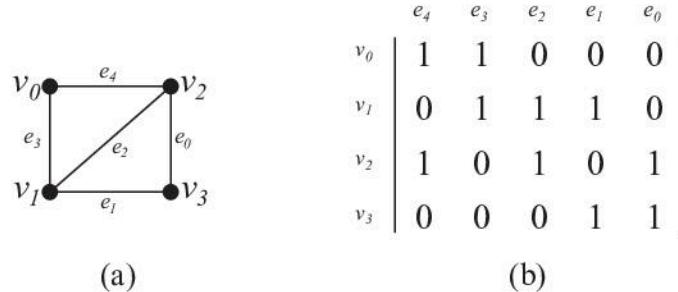
**Prova** Inicialmente, observamos que é imediato verificar que SUBCONJUNTO SOMA  $\in$  NP. Efetuaremos uma transformação a partir do problema COBERTURA POR VÉRTICES, que foi provado ser NP-completo através do Teorema 9.4. A entrada para este último é um grafo  $G(V, E)$  com  $|V| = n$ ,  $|E| = m$  e um inteiro  $k$ . A partir de  $G$  e  $k$ , construimos a entrada para SUBCONJUNTO SOMA, isto é, o conjunto  $S$  e o valor  $t$ . Para representar o grafo  $G$ , utilizamos a matriz de incidências  $B$  de  $G$ , definida no Capítulo 2. A matriz  $B$  possui uma linha para cada vértice  $v_i$  de  $G$ ,  $0 \leq i \leq n - 1$ , e uma coluna para cada aresta  $e_j$ ,  $m - 1 \leq j \leq 0$ . Por conveniência, na matriz  $B$  as colunas serão consideradas em ordem decrescente de valores de  $j$ , ou seja,  $j = m - 1, \dots, 0$ . A construção de  $S$  é a seguinte. Este conjunto terá  $n + m$  elementos, onde cada vértice e cada aresta de  $G$  contribuem com uma unidade. Os elementos de  $S$  correspondem à sequências binárias de  $m + 1$  dígitos. O dígito mais à esquerda está na posição  $j = m$ , e os demais seguem em ordem decrescente de  $j$ , até alcançar  $j = 0$ . Cada vértice  $v_i$  de  $G$  corresponderá à sequência binária  $p_i$ , a qual consiste no dígito 1, na posição  $j = m$ , seguido da linha  $i$  da matriz de incidências  $B$ , correspondente a  $v_i$ . Cada aresta  $e_j$  de  $G$  corresponderá a uma sequência binária  $q_j$ , consistindo no dígito 0, na posição  $j = m$ , seguido de um número binário que contém exatamente um valor 1, na posição  $j$ , e o valor 0 nas demais posições, para  $j = m - 1, \dots, 0$ .

O conjunto desses  $n + m$  valores binários pode ser disposto em uma matriz  $B'$ , de  $n + m$  linhas, uma para cada elemento de  $S$ , e  $m + 1$  colunas, de acordo com a descrição dada. Para representar cada um dos elementos de  $S$  utilizaremos, parcialmente, a base numérica 4, ao invés da base 2. Assim, cada elemento  $p_i$  de  $S$ , correspondente ao vértice  $v_i$  de  $G$ , será igual ao valor

$$p_i = \sum_{j=0}^{m-1} b_{ij} 4^j.$$

Observe que a parcela  $4^m$  de  $p_i$  corresponde ao dígito 1 que precede a linha  $i$  de  $B$ , na sequência binária relativa ao vértice  $v_i$ . Por outro lado, o elemento  $q_j$  de  $S$ , correspondente a aresta  $e_j$  de  $G$ , será igual a

$$q_j = 4^j,$$

Figura 9.18: Grafo  $G$  e sua matriz de incidências  $B$ .

	B					
$p_0$	1	1	1	0	0	$\rightarrow 4^5 + 4^4 + 4^3 = 1.344$
$p_1$	1	0	1	1	1	$\rightarrow 4^5 + 4^3 + 4^2 + 4^1 = 1.108$
$p_2$	1	1	0	1	0	$\rightarrow 4^5 + 4^4 + 4^2 + 4^0 = 1.297$
$p_3$	1	0	0	0	1	$\rightarrow 4^5 + 4^1 + 4^0 = 1.029$
$q_0$	0	0	0	0	0	$\rightarrow 4^0 = 1$
$q_1$	0	0	0	0	1	$\rightarrow 4^1 = 4$
$q_2$	0	0	0	1	0	$\rightarrow 4^2 = 16$
$q_3$	0	0	1	0	0	$\rightarrow 4^3 = 64$
$q_4$	0	1	0	0	0	$\rightarrow 4^4 = 256$

Figura 9.19: Matriz  $B'$ , dos elementos de  $S$ .

pois a representação binária correspondente a  $e_j$  contém um único valor 1, exatamente na posição  $j$ .

Resta ainda definir o valor  $t$ . Para tal, utilizaremos também uma representação com  $m + 1$  dígitos. O primeiro dígito é igual a  $k$ , e os demais são todos iguais a 2. Na representação numérica utilizada este valor corresponde a

$$t = k \cdot 4^m + \sum_{j=0}^{m-1} 2 \cdot 4^j.$$

A construção da entrada de SUBCONJUNTO SOMA está completa. Antes de prosseguir na prova do teorema iremos ilustrar um exemplo da construção de  $S$  e  $t$ , e da matriz obtida.

O exemplo consiste na entrada do problema COBERTURA POR VÉRTICES, que é o grafo  $G$ , com  $n = 4$  vértices e  $m = 5$  arestas, ilustrado na Figura 9.18(a), e o valor  $k = 2$ . A matriz de incidências  $B$ , correspondente a  $G$ , é apresentada na Figura 9.18(b). A Figura 9.19 apresenta a matriz  $B'$ , que contém as representações binárias dos valores dos elementos do conjunto  $S$ , bem como as representações decimais equivalentes. As 4 primeiras linhas de  $B'$  ilustram os valores dos elementos de  $S$  correspondentes aos vértices de  $G$ , e as 5 linhas seguintes contêm os valores dos elementos das arestas de  $G$ . Os valores decimais correspondentes aos elementos de  $S$  aparecem à direita da matriz  $B'$ . O valor  $t$  correspondente, pela construção, é igual a  $t = 2 \cdot 4^5 + \sum_{j=0}^{m-1} 2 \cdot 4^j$ , o que corresponde ao valor decimal 2.730.

Retornando à prova, após a construção de  $S$  e  $t$ , é necessário agora mostrar que  $G$  possui uma cobertura por vértices  $V' \subseteq V$  de tamanho  $\leq k$  se e somente se  $S$  contém um subconjunto  $S' \subseteq S$ , cuja soma de seus valores é exatamente  $t$ .

( $\Rightarrow$ ): Suponha que  $G$  possui uma cobertura por vértices

$$V' = \{v_{i_1}, \dots, v_{i_k}\}$$

de tamanho  $k$ .

Seja o subconjunto  $S' \subseteq S$ , definido como

$$S' = \{p_{i_1}, \dots, p_{i_k}\} \cup \{q_j | e_j \text{ é incidente a um único vértice de } V'\}.$$

Mostramos que  $\sum_{s \in S'} s = t = k \cdot 4^m + \sum_{j=0}^{m-1} 2 \cdot 4^j$ .

A primeira parcela ( $k \cdot 4^m$ ) de  $t$  é obtida somando os valores mais significativos da representação de  $\{x_{i_1}, \dots, x_{i_k}\}$ .

Para obter as demais parcelas de  $t$ , consideremos cada aresta  $e_j$  de  $G$ ,  $j = 0, \dots, m-1$ . Como  $V'$  é uma cobertura por vértices, cada  $e_j$  é incidente a pelo menos um vértice de  $V'$ . Isto é, para cada  $e_j$ , há pelo menos um valor  $v_{i_\ell}$ , tal que  $b_{i_\ell j} = 1$ .

**Caso 1:** A aresta  $e_j$  é incidente a dois vértices de  $V'$ . Então esses dois vértices contribuem com uma unidade cada para a soma  $t$ , na  $j$ -ésima posição. Por outro lado, a aresta  $e_j$  não contribui com qualquer valor, pois a definição de  $S'$  implica que  $e_j \notin S'$ . Então na posição  $j$  há uma contribuição de 2 unidades, isto é, a parcela  $2 \cdot 4^j$ , no valor de  $t$ .

**Caso 2:** A aresta  $e_j$  é adjacente a um único vértice  $v_{i_\ell}$ . Nesse caso,  $e_j \in S'$ . Então  $e_j$  contribui com uma unidade para a soma do  $j$ -ésimo dígito de  $t$ . Além disso,  $v_{i_\ell}$  contribui com uma unidade na soma de  $t$ . Isto é, a contribuição total é de duas unidades, ou seja, a parcela  $2 \cdot 4^j$ , no valor de  $t$ .

Desse modo, os elementos de  $S'$  somam exatamente  $t = k \cdot 4^m + \sum_{j=0}^{m-1} 2 \cdot 4^j$ .

( $\Leftarrow$ ): Suponha que existe  $S' \subseteq S$ , tal que  $\sum_{s \in S'} s = t$ .

Seja  $S' = \{p_{i_1}, \dots, p_{i_\ell}\} \cup \{q_{i_1}, \dots, q_{i_s}\}$ .

Vamos mostrar que  $V' = \{v_{i_1}, \dots, v_{i_\ell}\}$  é uma cobertura por vértices, de tamanho  $\ell = k$ .

Inicialmente, observamos que para cada aresta  $e_j \in E$ , existem três 1's no conjunto  $S$  correspondentes à  $j$ -ésima posição: um 1, para cada um dos dois vértices incidente a  $e_j$ , e um terceiro correspondente ao valor  $q_j$ . No esquema de representação da base 4 que utilizamos, não há “vai um” para o cálculo de  $t$ . Assim, para cada uma das posições de ordem  $< m$ , um ou dois valores de  $p_i$  contribuem para a soma de  $t$ . A contribuição de um desses valores já é suficiente para garantir que  $V'$  é uma cobertura por vértices de  $G$ . Finalmente, examinemos os 1's mais significativos, isto é, na posição  $m$ , de  $S'$ . A única maneira de obter os 1's mais significativos na expressão de  $t$  é através da inclusão de exatamente  $k$  valores  $p_i$ 's na soma. Isto é,  $\ell = k$ .

Em seguida à prova de NP-completude do problema SUBCONJUNTO SOMA, descrevemos, a seguir, um algoritmo para resolvê-lo. Utilizamos a técnica de programação dinâmica, descrita no Capítulo 5.

São dados um conjunto  $S$ , formado por  $n$  valores inteiros não negativos  $s_i$ , com possíveis repetições de valores, e dado um inteiro  $t > 0$ , determinar se  $S$  contém um subconjunto  $S' \subseteq S$ , tal que  $\sum_{s_i \in S'} s_i = t$ .

Empregamos uma matriz booleana  $X$ , de dimensão  $n \times (t+1)$ , com elementos  $x_{ij}$ , onde  $1 \leq i \leq n$  e  $0 \leq j \leq t$ . A ideia é que  $x_{ij}$  seja verdadeiro (V) ou falso (F), dependendo das seguintes condições

$$i = 1 \Rightarrow x_{ij} = \begin{cases} V, & \text{se } j = 0 \text{ ou } j = s_1 \\ F, & \text{caso contrário} \end{cases}$$

$$i > 1 \Rightarrow x_{ij} = \begin{cases} V, & \text{se } x_{i-1,j} = V, \text{ ou } j - s_i \geq 0 \text{ e } x_{i-1,j-s_i} = V \\ F, & \text{caso contrário} \end{cases}$$

Então, a resposta do problema SUBCONJUNTO SOMA será SIM se  $x_{n,t} = V$ , e NÃO caso contrário.

A justificativa de correção deste algoritmo se baseia na seguinte observação.

Para  $j \neq 0$ , sempre que  $x_{i,j} = V$ , existe um subconjunto  $S_i \subseteq \{s_q, \dots, s_i\}$  tal que  $\sum_{s_i \in S_i} s_i = j$ .

A condição apresentada é trivialmente satisfeita para  $i = 1$ , de acordo com a definição de  $x_{1,j}$ . Note  $x_{1,j} = V$ , pois nesse caso o subconjunto  $S_1 = \emptyset$  satisfaz a condição. Para  $i > 1$ , é imediato constatar que a correção da observação anterior é preservada.

O Algoritmo 9.1, seguinte, descreve o processo.

**Algoritmo 9.1:** Subconjunto Soma

**Dados:** Conjunto  $S = \{s_1, \dots, s_n\}$ ,  $n > 1$  cada  $s_i$  inteiro não negativo e um valor inteiro  $t$

$X :=$  matriz booleana, elementos  $x_{ij}$ ,  $1 \leq i \leq n$  e  $0 \leq j \leq t$

**para**  $j = 0, \dots, t$  efetuar

**se**  $j = 0$  ou  $j = s_1$  **então**

$x_{1j} = V$

**caso contrário**

$x_{1j} = F$

**para**  $i = 2, \dots, n$  efetuar

**para**  $j = 0, \dots, t$  efetuar

**se**  $x_{i-1,j} = V$ , ou  $j - s_i \geq 0$  e  $x_{i-1,j-s_i} = V$  **então**

$x_{ij} = V$

**caso contrário**

$x_{ij} = F$

**se**  $x_{nt} = V$  **então**

        Resposta SIM

**caso contrário**

        Resposta NÃO

$i \backslash j$	0	1	2	3	4	5	6	7	8
1	$V$	$V$	$F$						
2	$V$	$V$	$V$	$V$	$F$	$F$	$F$	$F$	$F$
3	$V$	$F$							
4	$V$								
5	$V$								

Figura 9.20: Matriz  $X$  do exemplo

Como exemplo de aplicação do algoritmo, seja o conjunto  $S = \{1, 2, 4, 5, 6\}$ , e o valor  $t = 8$ .

A matriz  $X$  construída pelo algoritmo é apresentada na Figura 9.20:

Portanto, como  $X_{58} = V$ , concluimos que  $S$  contém um subconjunto, cuja soma de seus elementos é igual a  $t$ . De fato, o subconjunto  $S' = \{1, 2, 5\}$  satisfaz.

Para cumprir o objetivo desejado, é necessário que o progresso da computação se desenvolva para valores crescentes de  $i$ , iniciando-se em  $i = 1$  e terminando com  $i = n$ , conforme estabelecido no algoritmo.

Nesse caso, cada valor booleano  $x_{ij}$ ,  $V$  ou  $F$ , pode ser determinado em tempo constante. Ou seja, a complexidade do algoritmo é igual ao número de elementos da matriz, isto é,  $O(nt)$ . Por outro lado, o tamanho da entrada é  $O(n \log t)$ , pois qualquer elemento de  $S$  que seja maior do que  $t$  pode ser ignorado no processo. Contudo,  $t$  não é necessariamente polinomial em  $n$ , pois o valor de  $t$  e os valores de  $s_i \in S$  podem ser arbitrariamente grandes. Portanto, não se trata de um algoritmo polinomial. Contudo, este algoritmo seria polinomial se a codificação dos dados fosse em unário, pois nesse caso o tamanho da entrada seria proporcional a  $t$ . Então trata-se de um algoritmo pseudopolinomial.

**Corolário 9.1**

O problema SUBCONJUNTO SOMA é NP-completo fraco.

## 9.13 Programas em Python

Esta seção contém implementações dos algoritmos formulados neste capítulo. As implementações seguem as descrições gerais apresentadas nos Capítulos 1 e 2.

### 9.13.1 Algoritmo 9.1: Subconjunto Soma

O Algoritmo 9.1 é implementado como tradução direta para a linguagem.

Programa 9.1: Subconjunto Soma

```

1 #Algoritmo 9.1: Subconjunto Soma
2 #Dados: Conjunto S = {s1,...,sn}, n>1, cada s[i] inteiro não negativo e um valor inteiro t
3 def SubconjuntoSoma(S,n,t):
4     S = [None] + S
5     x = [None]*(n+1)
6     for i in range(1,n+1):
7         x[i] = [None]*(t+1)
8     for j in range(t+1):
9         x[1][j] = (j==0 or j==S[1])
10    for i in range(2,n+1):
11        for j in range(t+1):
12            x[i][j] = x[i-1][j] or (j-S[i] >= 0 and x[i-1][j-S[i]])
13    return x[n][t]

```

## 9.14 Exercícios

9.1 Sejam  $A, B$  dois problemas tais que  $A \in \text{NP}$  e  $B \notin \text{NP}$ . Então  $B$  é polinomialmente transformável em  $A$  se e somente se  $\text{P} = \text{NP}$ . Certo ou errado?

9.2 Se  $\text{P} = \text{NP}$  todo problema NP-difícil é polinomial. Certo ou errado?

9.3 Provar que o problema CICLO HAMILTONIANO é NP-completo.

9.4 Usando o resultado do exercício anterior, provar que o problema de decisão CAIXEIRO VIAJANTE é NP-completo.

9.5 Considere o problema CAMINHO MÍNIMO CONDICIONADO, definido a seguir:

**DADOS:** Um grafo  $G(V, E)$ , onde cada aresta possui um peso inteiro positivo; um par de vértices  $s, t \in V$ ; um subconjunto  $V' \subseteq V$ ; um inteiro  $k > 0$

**DECISÃO:** Existe um caminho  $P$  entre  $s$  e  $t$  em  $G$ , tal que  $P$  contém todos os vértices de  $V'$  e a soma dos pesos das arestas de  $P$  é  $< k$ ?

Provar Que esse problema é NP-completo.

9.6 Provar que o problema CAMINHO MÍNIMO CONDICIONADO do Exercício 9.5

- (i) permanece NP-completo quando o grafo  $G$  é substituído por um digrafo  $D$ , mas, por outro lado
- (ii) torna-se polinomial quando  $D$  é um digrafo acíclico (nesse caso, apresentar um algoritmo polinomial para resolver o problema e determinar a sua complexidade).

- 9.7 O problema CAMINHO MÍNIMO CONDICIONADO do 9.5 permanece NP-completo quando os pesos das arestas do grafo são todos unitários. Provar ou apresentar algoritmo polinomial.
- 9.8 Provar que o problema de determinar a árvore geradora da altura máxima de um grafo  $G$  é NP-difícil (e a de altura mínima?).
- 9.9 Sejam dados um grafo  $G$  e um inteiro  $k > 0$ . Provar que o problema de determinar uma árvore geradora  $T$  de  $G$ , tal que cada vértice possui grau  $\leq k$ , em  $T$ , é NP-completo.
- 9.10 Sejam  $G$  um grafo conexo e  $k > 0$  um inteiro. O problema de determinar uma árvore de profundidade de altura  $\geq k$  em  $G$  é uma restrição do problema de encontrar em  $G$  uma árvore geradora (qualquer) de altura  $\geq k$ . Certo ou errado?
- 9.11 Seja  $A$  um problema polinomialmente transformável em  $B$ . Se  $B$  admite algoritmo pseudopolinomial para resolvé-lo,  $A$  também o admite. Certo ou errado?
- 9.12 Modificar o Algoritmo 9.1, que resolve o problema SUBCONJUNTO SOMA, de modo a obter também um subconjunto cuja soma de seus elementos seja igual a  $t$ , em caso de resposta positiva. A complexidade do algoritmo não deve ser alterada.
- 9.13 Se o complemento  $\overline{\Pi}$  de todo problema  $\Pi \in \text{NP}$  for tal que  $\overline{\Pi}$  também esteja em NP, então:
- (i)  $\text{NP} = \text{Co-NP}$
  - (ii)  $\text{P} = \text{NP}$
- Certo ou errado?
- 9.14 Um problema de decisão  $\Pi$  é *Co-NP-completo*, quando:
- (i)  $\Pi \in \text{Co-NP}$ , e
  - (ii) todo problema de decisão  $\Pi' \in \text{Co-NP}$  satisfaz  $\Pi' \propto \Pi$ .
- Então as classes NP-completo e Co-NP-completo são disjuntas se e somente se  $\text{P} \neq \text{NP}$ .
- Certo ou errado?
- 9.15 ENADE 2014
- Um cientista afirma ter encontrado uma redução polinomial de um problema NP-Completo para um problema pertencente à classe P. Considerando que essa afirmação tem implicações importantes no que diz respeito à complexidade computacional, avalie as seguintes asserções e a relação proposta entre elas.

I A descoberta do cientista implica  $P = NP$

PORQUE

II A descoberta do cientista implica a existência de algoritmos polinomiais para todos os problemas NP-Completos.

A respeito dessas asserções, assinale a opção correta.

- a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa correta da I
- c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa
- d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

#### 9.16 ENADE 2014

Considere o processo de fabricação de um produto siderúrgico que necessita passar por  $n$  tratamentos térmicos e químicos para ficar pronto. Cada uma das  $n$  etapas de tratamento é realizada uma única vez na mesma caldeira. Além do custo próprio de cada etapa do tratamento, existe o custo de se passar de uma etapa para a outra, uma vez que, dependendo da sequência escolhida, pode ser necessário alterar a temperatura da caldeira e limpá-la para evitar a reação entre os produtos químicos utilizados. Assuma que o processo de fabricação inicia e termina com a caldeira limpa. Deseja-se projetar um algoritmo para indicar a sequência de tratamentos que possibilite fabricar o produto com o menor custo total.

Nessa situação conclui-se que:

- a) a solução do problema é obtida em tempo de ordem  $O(n \log n)$ , utilizando um algoritmo ótimo de ordenação.
- b) uma heurística para a solução do problema de coloração de grafos solucionará o problema em tempo polinomial.
- c) o problema se reduz a encontrar uma árvore geradora mínima para o conjunto de etapas do processo, requerendo tempo de ordem polinomial para ser solucionado.
- d) a utilização do algoritmo de Dijkstra para se determinar o caminho de custo mínimo entre o estado inicial e o final soluciona o problema em tempo polinomial.
- e) qualquer algoritmo conhecido para a solução do problema descrito possui ordem de complexidade de tempo não polinomial, uma vez que o problema do caixeiro viajante se reduz a ele.

#### 9.17 ENADE 2005

A análise de complexidade provê critérios para a classificação de problemas com base na computabilidade de suas soluções, utilizando-se a máquina de Turing como modelo referencial e possibilitando o agrupamento de problemas em classes.

Nesse contexto, julgue os itens a seguir.

I É possível demonstrar que  $P \subseteq NP$  e  $NP \subseteq P$ .

II É possível demonstrar que se  $P \neq NP$ , então  $P \cap NP\text{-Completo} = \emptyset$ .

- III Se um problema  $Q$  é NP-difícil e  $Q \in \text{NP}$ , então  $Q$  é NP-completo.
- IV O problema da satisfatibilidade de uma fórmula booleana  $F$  foi provado ser NP-difícil e NP-Completo.
- V Encontrar o caminho mais curto entre dois vértices dados em um grafo de  $n$  vértices e  $m$  arestas não é um problema da classe P.

Estão certos apenas os itens

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

#### 9.18 UVA Online Judge 11407

Faça um programa para o seguinte problema:

Qualquer inteiro positivo  $n$  pode ser escrito como  $n = a_1^2 + a_2^2 + \dots + a_m^2$ , isto é, qualquer número inteiro positivo pode ser representado como soma de quadrados inteiros menores. Sua tarefa é encontrar o menor ‘ $m$ ’ tal que  $n = a_1^2 + a_2^2 + \dots + a_m^2$ .

## 9.15 Notas Bibliográficas

Os trabalhos pioneiros na teoria do NP-completo são Cook (1971) e Karp (1972). Foi Cook (1971) quem introduziu a teoria do NP-completo e provou a existência do primeiro problema NP-completo (Teorema 9.1). Um livro dedicado ao assunto é Garey e Johnson (1979), o qual contém também uma lista com centenas de problemas NP-completos. Partes da teoria são apresentadas também em capítulos de livros como Aho, Hopcroft e Ullman (1974), Baase (1978), Even (1979), Horowitz e Sahni (1978), Papadimitriou e Steiglitz (1982), Reingold, Nievergelt e Deo (1977). Os dez problemas da Seção 9.4 foram todos retirados dos citados trabalhos pioneiros a saber: SATISFATIBILIDADE, CONJUNTO INDEPENDENTE DE VÉRTICES, CLIQUE e ISOMORFISMO DE SUBGRAFOS de Cook (1971), enquanto que os demais podem ser encontrados em Karp (1972). A terminologia corrente, utilizada na teoria, é principalmente de Karp (1972). Os conceitos de NP-completo forte e fraco são de Garey e Johnson (1978). Uma terminologia diferente para esses últimos foi proposta por Lageweg, Lawler, Lenstra e Rinnooy Kan (1978). Knuth (1974) discute a terminologia do NP-completo, com uma dose de humor. A teoria foi estabelecida, de forma independente, por Levin (1973). Exemplos de problemas comprovadamente intratáveis aparecem em Hopcroft e Ullman (1979). O critério de considerar os algoritmos polinomiais como eficientes é anterior a 1971. Com efeito, esse critério havia sido mencionado por Cobham (1964) e Edmonds (1965). Informações posteriores sobre o que ocorreu na área do NP-completo foram publicadas por Jonhson, na coluna de NP-completude do Journal of Algorithms, entre 1981 e 1985. Uma referência clássica e das mais relevantes para a teoria de NP-completude é o livro Papadimitriou (1994). Por outro lado, o texto de Arora e Barak (2009), mais recente, é outra indicação. Finalmente, o livro de Moore e Martens (2009) apresenta diversos conceitos considerados mais avançados de maneira acessível. O algoritmo que verifica a primalidade de inteiros é conhecido como Algoritmo AKS de autoria de Agrawal, Kayal e Saxena (2004). Apesar de que o Algoritmo AKS verifica em tempo polinomial se um dado número  $n$  é primo, a questão da existência ou não de um algoritmo polinomial para decompor  $n$  em fatores primos constitui um importante problema em aberto.

---

# Referências

---

- Agrawal, M., Kayal, K., Saxena, N. (2004) Primes is in P. *Annals of Mathematics*, 160, 781-793.
- Aho, A.V., Hopcroft J.E., Ullman, J.D. (1974) *The Design and Analysis of Computer Algorithms*. Reading: Addison-Wesley.
- Andrade, M.C.Q. (1980) *A Criação no Processo Decisório*. Rio de Janeiro: LTC.
- Appel, K., Haken, W. (1977a) Every planar map is four colorable, part I: Discharging. *Illinois J. of Math*, 21, 429-490.
- Appel, K., Haken, W. (1977b) Every planar map is four colorable, part II: Reducibility. *Illinois J. of Math*, 21, 491-567.
- Arora, S., Barak, B. (2009) *Computational Complexity: A Modern Approach*. Cambridge: Cambridge University Press.
- Ascencio, A.F.G., Araujo, G.S. (2011) *Estruturas de Dados: Algoritmos, Análise da Complexidade e Implementação em Java e C/C++*. São Paulo: Pearson.
- Baase, S. (1978) *Computer Algorithms: Introduction to Design and Analysis*. Reading: Addison-Wesley.
- Barbosa, R.M. (1974) *Combinatória e Grafos*. São Paulo: Nobel.
- Barron, D.W. (1968) *Recursive Techniques in Programming*. MacDonald, Londres, Nova York: Elsevier.
- Bellman, R.E. (1957) *Dynamic Programming*. Princeton: Princeton University Press.
- Bellman, R.E. (1958) On a routing problem. *Quart. Applied Mathematics*, 16, 87-90.
- Berge, C. (1957) Two Theorems in Graph Theory. *Proc. Natl. Acad. Sci.* 43, 842-844.
- Berge, C. (1962) *The Theory of Graphs and its Applications*. Nova York: John Wiley.
- Berge, C. (1973) *Graphs and Hypergraphs*. Amsterdam: North-Holland.
- Berge, C. (1985) *Graphs*. Amsterdam: North-Holland.
- Berztiss, A.T. (1971) *Data Structures: Theory and Practice*. Nova York: Academic Press.
- Biggs, N.L., Lloyd, E.K., Wilson, R.J. (1976) *Graph Theory 1736-1936*. Oxford: Clarendon Press.
- Blum, N. (2015) Maximum Matching in General Graphs Without Explicit Consideration of Blossoms Revisited, *arXiv:1509.04927 [cs.DS]*.
- Boaventura, P.O. (2012) *Grafos: Teoria, Modelos, Algoritmos*. 2<sup>a</sup> ed. São Paulo: Edgar Blucher Ltda.
- Bollobás, B. (1978) *Extremal Graph Theory*. Londres: Academic Press.
- Bollobás, B. (1979) *Graph Theory: An Introductory Course*. Nova York: Springer-Verlag.
- Bollobás, B. (1998) *Modern Graph Theory*. Nova York: Springer.
- Bondy, J.A., Murty, U.S.R. (1976) *Graph Theory with Applications*. Nova York: North-Holland.
- Bondy, J.A., Murty, U.S.R. (2008) *Graph Theory*. Nova York: Springer.
- Boruvka, O. (1926) Contribution to the solution of a problem of economical construction of electrical networks. *Elektronik Obzor* 15, 153-154 (em Tcheco).
- Brandstädt, A., Le, V.B., Spinrad, J.P. (1999) *Graph Classes: A Survey*. SIAM.
- Brelaz, D. (1979) New methods to color the vertices of a graph. *Comm of the ACM*, 22, 251-256.
- Busacker, R.G., Saaty, T.L. (1965) *Finite Graphs and Networks: An Introduction with Applications*. Nova York: McGraw-Hill.
- Capobianco, M., Molluzzo, J.C. (1978) *Examples and Counter Examples in Graph Theory*. Nova York: North-Holland.

- Cardoso, D.M., Szymanski, J., Rostami, M. (2009) *Matemática Discreta: Combinatória, Teoria dos Grafos e Algoritmos*. Lisboa: Escolar Editora, Lisboa.
- Carré, B. (1979) *Graphs and Networks*. Oxford: Clarendon Press.
- Carvalho, R.L. (1981) *Máquinas, Programas e Algoritmos*. 2<sup>a</sup> Escola de Computação, Instituto de Matemática, Estatística e Ciência da Computação, Universidade de Campinas, Campinas.
- Cayley, A. (1857) On the theory of analytical forms called trees. *Phil. Mag.*, 13, 172-176.
- Cayley, A. (1889) A theorem on trees. *Quart. J. Math.*, 23, 376-378.
- Chazelle, B. (2000) A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM* 47, 6, 1028-1047.
- Cheriton, D., Tarjan, R.E. (1976) Finding minimum spanning trees. *SIAM J. Comp.*, 5, 724-742.
- Cherkassky, B. (1977) Algorithm of construction of maximal flow in networks with complexity  $O(|V|^2\sqrt{E})$  operations, *Math. Methods of Solution of Economic Problems*, 7, 117-125.
- Cherkassky, B.V., Goldberg, A.V., Radzik, T. (1994) Shortest Paths Algorithms: Theory and Experimental Evaluation. *Proc. 5<sup>th</sup> ACM-SIAM Symposium on Discrete Algorithms*, 515-525.
- Christofides, N. (1975) *Graph Theory: An Algorithmic Approach*. Nova York: Academic Press.
- Cobham, A. (1964) The Intrinsic Computational Difficulty of Functions. In: Bar-Hillel, Y. (ed.). *Proc. 1964 International Congress for Logic Methodology and Philosophy of Science*. Amsterdam: North-Holland, 24-30.
- Comeil, D.G., Graham, B. (1973) An algorithm for determining the chromatic number of a graph. *SIAM J. on Computing*, 2, 311-318.
- Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E. (2001) *Introduction to Algorithms*. 2<sup>a</sup> ed. McGraw-Hill Higher Education.
- Cook, S.A. (1971) The Complexity of Theorem-Proving Procedures. *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, New York, 151-158.
- Corneil, D.G. (2004) Lexicographic Breadth First Search: A Survey, Graph Theoretic Concepts in Computer Science. *30<sup>th</sup> International Workshop, WG 2004, Lecture Notes in Computer Science*, 3353. Springer-Verlag, 1-19.
- Deo, N. (1974) *Graph Theory with Applications to Engineering and Computer Science*, Englewood Cliffs: Prentice-Hall.
- Diestel, R. (1997) *Graph Theory*. Springer.
- Diestel, R. (1990) *Graph Decompositions: A Study in Infinite Graph Theory*. Clarendon Press.
- Dijkstra, E.W. (1959) A note on two problems in connection with graphs. *Numer. Math.*, 1, 269-271.
- Dilworth, R.P. (1950) A decomposition theorem for partially ordered sets. *Ann. Math.*, 51, 161-166.
- Dinic, E.A. (1970) Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11, 1277-1280.
- Diverio, T.A., Menezes, P.B. (2011) *Teoria da Computação: Máquinas Universais e Computabilidade*. 3<sup>a</sup> ed. Porto Alegre: Editora Bookman, Luzzeto.
- Dirac, G.A. (1952) Some theorems on abstract graphs. *Proc. London Math. Soc.*, 2, 69-81.
- Dirac, G.A. (1952a) A property of 4-chromatic graphs and some remarks on critical graphs. *J. London Math. Soc.*, 27, 85-92.
- Dirac, G.A. (1960) 4-chrome Graphen und vollständige 4-Graphen. *Math. Nachr.*, 22, 51-60.
- Dirac, G.A. (1961) On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25, 71-76.
- Dreyfus, S.E. (1969) An appraisal of some shortest path algorithms, *Operations Research*, 17, 395-412.
- Duan, R., Su, H.H. (2012) A Scaling Algorithm for Maximum Weight Matching in Bipartite Graphs. *Proc of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1413-1424.
- Edmonds, J. (1965) Minimum partition of a matroid into independent subsets, *J. Res. Nat. Bur. Standards Sect. B*, 69, 67-72.
- Edmonds, J., Karp, R.M. (1972) Theoretical improvements in algorithmic efficiency for network flow problems. *J. of the ACM*, 19, 248-264.
- Eppstein, D. (1998) Finding the k shortest paths. *SIAM J. Computing*, 28, 652-673.
- Euler, L. (1736) Solutio problematis ad geometriam situs pertinentis, *Academiae Petropolitanae*, 8, 128-140.
- Even, S. (1973) *Algorithmic Combinatorics*. Nova York: MacMillan.
- Even, S. (1979) *Graph Algorithms*. Potomac: Computer Science Press.

- Even, S., Kariv, O. (1975) An Algorithm for Maximum Matching in Graphs. *Proceedings of the 16<sup>th</sup> Annual IEEE Symposium on Foundations of Computer Science*. Nova York: IEEE, 100-112.
- Even, S., Tarjan, R.E. (1975) Network flow and testing graph connectivity. *SIAM J. on Camp.*, 4, 507-518.
- Fáry, I. (1948) On straight line representation of planar graphs. *Acta Sci. Math. Szeged*, 11, 229-233.
- Feofiloff, P., Lucchesi, C.L. (1988) *Algoritmos para Igualdades min-max em Grafos*, VI Escola de Computação.
- Fernandes, C.G., Miyazawa, F.K., Cerioli, M.R., Feofiloff, P. (2001) *Uma Introdução Sucinta a Algoritmos de Aproximação*. Colóquio Brasileiro de Matemática, IMPA.
- Figueiredo, C.M.H., Lamb, L.C. (2016) *Teoria da Computação: Uma Introdução à Complexidade e à Lógica Computacional*. Anais de 35a. Jornada de Atualização e Informática. Congresso da Sociedade Brasileira de Computação.
- Figueiredo, C.M.H., Lemos, H., Fonseca, G.D. e Sá, V.G.P. (2007) *Introdução aos Algoritmos Randomizados*. 36º Colóquio Brasileiro de Matemática, IMPA.
- Figueiredo, C.M.H., Meidanis, J., Mello, C.P. (1995) A linear-time algorithm for proper interval graph recognition. *Information Processing Letters*, 56, 179-184.
- Figueiredo, C.M.H., Szwarcfiter, J.L. (1999) *Emparelhamentos em Grafos: Algoritmos e Complexidade*. Jornada de Atualização em Informática, XXIII Congresso da Sociedade Brasileira de Computação, PUC-RJ.
- Floyd, R.W. (1957) Non deterministic algorithms, *J. of the ACM*, 14, 636-644.
- Floyd, R.W. (1962) Algorithm 97: shortest path. *Comm. of the ACM*, 5, 345.
- Ford, L.R. (1956) *Network Flow Theory*. Tech. Report P-923, RAND Corporation.
- Ford, L.R., Fulkerson, D.R. (1956) Maximal flow through a network, *Canadian J. Math.*, 8, 399-404.
- Ford, L.R., Fulkerson, D.R. (1957) A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. *Canadian J. Math.*, 9, 210-218.
- Ford, L.R., Fulkerson, D.R. (1962) *Flows in Networks*. Princeton: Princeton University Press.
- Fraysseix, H., Ossona de Mendez, P., Rosenstieh, P. (2006) Tremaux trees and planarity. *International Journal of Foundations of Computer Science*, 17, 1017-1030.
- Fritsch, R., Fritsch, E. (1998) *The Four-Colour Theorem*. Springer.
- Fulkerson, D.R., Gross, O.A. (1965) Incidence matrices and interval graphs. *Pacific J. Math.*, 15, 835-855.
- Furtado, A.L. (1973) *Teoria dos Grafos: Algoritmos*. Rio de Janeiro: LTC.
- Furtado, A.L., Santos, C.S., Veloso, P.A., Azevedo, P.A. (1982) *Estrutura de Dados*. Rio de Janeiro: Elsevier.
- Gabow H.N. (1976) An efficient implementation of Edmond's algorithm for maximum matching on graphs. *J. of ACM*, 221-234.
- Galil, Z. (1986) Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18, 23-38.
- Galil, Z. (1978) A New Algorithm for the Maximal Flow Problem. *Proc. 19<sup>th</sup> Symp. on the Foundations of Computer Science*, IEEE, 231-245.
- Galil, Z., Naamad, A. (1979) Network Flow and Generalized Path Compression, *Proc. 11<sup>th</sup> Ann. Symp. on Theory of Computing*, ACM, 13-26, 1979.
- Gallai, T. (1964) Elementare relationen bezüglich der gliederndtrennenden punkte von graphen, *Magyar Tud. Akad. Mat. Kutato Int. Kozl.*, 9, 235-236.
- Gallai, T. (1967), Transitiv orientierbare graphen, *Acta Math. Acad. Sc. Hungar.*, 18, 25-66.
- Garey, M.R., Johnson, D.S. (1976) The complexity of near-optimal graph coloring. *J. of the ACM*, 23, 43-49.
- Garey, M.R., Johnson, D.S. (1978) Strong NP-completeness results: motivation, examples and implications. *J. of the ACM*, 25 (1978), 439-508.
- Garey, M.R., Johnson, D.S. (1979), *Computers and Intractability: A Guide to the Theory of NP-completeness*. San Francisco: Freeman.
- Gerards, A.M.H. (1995) Matching. In: Bli, M.O., Magnanti, T.L., Monmaand, C.L., Nemhauser, G.L. (eds.) *Network Models, Handbooks in Operations Research and Management Science*, Vol. 7, 135-224.
- Ghouila-Houri, A. (1962) Caractérisation des graphes non orientés dont on peut orienter les arêtes de manière à obtenir le graphe d'une relation d'ordre. *C. R. Acad. Sc. Paris*, 254, 1370-1371.
- Gilmore, P.C., Hoffman, A.J. (1964) A characterization of comparability graphs and of interval graphs. *Canad. J. of Math.*, 16, 539-548.

- Goldberg, A., Tardos, E., Tarjan, R.E. (1990) *Network Flow Algorithms in Paths, Flows and VLSI-Layout*. Korte et al. (eds.) Springer Verlag.
- Goldberg, A. (1986) *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD Thesis, Department of Electrical Engineering and Computer Science, MIT.
- Goldberg, A., Tarjan, R.E. (1986) A New Approach to the Maximum Flow Problem. *Proc. 18<sup>th</sup> ACM Symposium on Theory of Computing*, 136-146.
- Goldbach, M., Goldbach, E. (2012) *Grafos: Conceitos, Algoritmos e Aplicações*. Elsevier.
- Golomb, S.W., Baumert, L.D. (1965) Backtrack programming. *J. of the ACM*, 12, 516-524.
- Golumbic, M.C. (2004) *Algorithmic Graph Theory and Perfect Graphs*. 2<sup>a</sup> ed. Nova York: North Rolland.
- Golumbic, M.C., Trenk, A.N. (2004) *Tolerance Graphs*. Cambridge: Cambridge University Press.
- Gondran, M., Minoux M. (1979) *Graphes et Algorithmes*. Paris: Editions Eyrolles.
- Goodman, S.E., Hedetniemi, S.T. (1977) *Introduction to Design and Analysis of Algorithms*. Nova York: McGraw-Hill.
- Greene, D.H., Knuth, D.E. (1981) *Mathematics for the Analysis of Algorithms*. Boston: Birkhauser, 1981.
- Habib, M., McConnell, R., Paul, C., Viennot, L. (2000) Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234, 59-84.
- Hadlock, F.O. (1974) Minimum Spanning Forests of Bounded Trees, *Proc. 5<sup>th</sup> Southeastern Conference on Combinatorics, Graph Theory and Computing*. Winnipeg: Utilitas Mathematica Pub. 449-460.
- Hall, P. (1935) On representations of subsets. *J. London Math. Soc.*, 10, 26-30.
- Hall, M. (1956) An algorithm for distinct representations. *Amer. Math. Monthly*, 63, 716-717.
- Halldórsson, M.M. (1993) Still better performance guarantee for approximate graph coloring. *Information Processing Letters*, 45, 19-23.
- Harary, F. (1969) *Graph Theory*. Reading: Addison-Wesley.
- Harary, F., Norman, R.Z., Cartwright, D. (1965) *Structural Models: An Introduction to the Theory of Directed Graphs*, John. Nova York: Wiley.
- Harary, F., Prins, G. (1966) The block-culpoint-tree of a graph. *Publ. Math. Debrecen*, 13, 103-107.
- Harrison, M.C. (1973) *Data Structures and Programming*. Glenview: Scott, Foresman & Co.
- Heawood, P.J. (1890) Map colour theorems. *Quart. J. Math.*, 24, 332-338.
- Henzinger, M.R., Klein, P., Rao, S., Subramanian, S. (1997) Faster shortest-path algorithms for planar graphs. *J of Computer and System Sci.*, 55, 3-23.
- Hopcroft, J., Karp, R.M. (1973) An n5/2 algorithm for maximum matchings in bipartite graphs. *SIAM J. Comp.*, 2, 225-231.
- Hopcroft, J., Tarjan, R. (1973a) Dividing a graph into tri-connected components. *SIAM J. Comp.*, 2, 135-158.
- Hopcroft, J., Tarjan, R. (1973b) Efficient algorithms for graph manipulation. *Comm. of the ACM*, 16, 372-378.
- Hopcroft, J. and Tarjan, R. (1974), Efficient planarity testing, *J. of the ACM*, 21 (1974), 549-568.
- Hopcroft, J.E., Ullman, J.D. (1969) *Formal Languages and their Relation to Automata*. Reading: Addison-Wesley.
- Hopcroft, J.E., Ullman, J.D. (1979). *Introduction to Automata Theory, Languages and Computation*. Reading: Addison-Wesley.
- Horowitz, E., Sahni, S. (1976) *Fundamentals of Data Structures*. Woodland Hills: Computer Science Press.
- Horowitz, E., Sahni, S. (1978) *Fundamentals of Computer Algorithms*. Rockville: Computer Science Press.
- Hu, T.C. (1982) *Combinatorial Algorithms*. Reading: Addison-Wesley.
- Itai, A., Shiloach, Y. (1979) Maximum flow in planar networks. *SIAM J. on Comp.*, 8, 135-150.
- Jensen, T.R., Toft, B. (1995) *Graph Coloring Problems*. Nova York: Wiley-interscience.
- Johnson, D.B. (1975) Finding all the elementary circuits of a directed graph. *SIAM J. Comp.*, 4, 77-84.
- Johnson, D.S. (1986) The NP-completeness column: an ongoing guide, *Journal of Algorithms*, 7, 584-601.
- Jordan, C. (1869) Sur les assemblages de lignes. *J. Reine Angew Math.*, 70, 185-190.
- Kahn, A.B. (1962) Topological sorting for large networks. *Comm. of the ACM*, 5, 558-562.
- Karp, R.M. (1972) Reducibility Among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W. (eds.) *Proc. of a Symposium on the Complexity of Computer Computations*. Nova York: Plenum Press.
- Karp, R.M. (1975) On the complexity of combinatorial problems, *Networks*, 5, 45-68.

- Karzanov, A.V. (1974) Determining the max flow in a network by the method of pre-flows, *Soviet Math. Dokl.*, 15, 434-437.
- Kase, R.H. (1963) Topological sorting for PERT networks. *Comm. of the ACM*, 6, 738-739.
- Kempe, A.B. (1879) On the geographical problem of four colors. *Amer. J. Math.*, 2, 193-204.
- Kirchhoff, G. (1847) Über die Auflösung der Gleichungen, auf welche man bei der Untersuchungen der linearen Verteilung galvanischer Ströme geführt wird. *Ann. Phys. Chem.*, 72, 497-508.
- Kleinberg, J., Tardos, E. (2005) *Algorithm Design*. Boston: Addison-Wesley Longman Publishing Co.
- Knuth, D.E. (1997) *The Art of Computer Programming, volume I: Fundamental Algorithms*. 3<sup>a</sup> ed. Reading: Addison-Wesley.
- Knuth, D.E. (1998) *The Art of Computer Programming, volume III: Sorting and Searching*. Reading: Addison-Wesley.
- Knuth, D.E. (1974a) Structured programming with go to statements. *Comp. Surveys*, 6, 26-30.
- Knuth, D.E. (1974b) A terminological proposal. *SIGACT News*, 6, 12-18.
- Knuth, D.E. (1975) Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29, 121-136.
- Kuhn, H.W. (1955) The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 83-97.
- König, D. (1936) *Theorie der Endlichen und Unendlichen Graphen*. Leipzig.
- Kruskal Jr., J.B. (1956) On the shortest spanning subtrees of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7, 48-50.
- Kuratowski, K. (1930) Sur le problème des courbes gauches en topologies. *Fund. Math.*, 15, 271-283.
- Lageweg, B.J., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. (1978) *Computer Aided Complexity Classification of Deterministic Scheduling Problems*. Amsterdam: Mathematisch Centrum.
- Lasser, D.J. (1961) Topological ordering of a list of randomly numbered elements of a network. *Comm. of the ACM*, 4, 167-168.
- Lawler, E.L. (1972) A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Sci.*, 18, 401-405.
- Lewis, H.R., Papadimitriou (2007) *Elements of the Theory of Computation*. 2<sup>a</sup> ed. Nova York: Prentice-Hall.
- Levin, L. A. (1973) Universal sorting problems, *Problemy Peredacy Informacii*, 9, 115-116.
- Lovász, L. (1979) *Combinatorial Problems and Exercises*. Amsterdam: North-Holland.
- Lovász, L., Plummer, M.D. (2009) *Matching Theory*. American Mathematical Society, Chelsea Publishing.
- Lucas, E. (1882) *Récreations Mathématiques*. Paris.
- Lucchesi, C.L. (1979) *Introdução à Teoria de Grafos*. 12º Colóquio Brasileiro de Matemática, Poços de Caldas.
- Lucchesi, C.L., Simon, I., Simon, J., Kowaltowski, T. (1979) *Aspectos Teóricos de Computação*. Instituto de Matemática Pura e Aplicada (Projeto Euclides), Rio de Janeiro.
- Lucena, C.J.P. (1972) *Introdução às Estruturas de Informação*. Rio de Janeiro: LTC.
- Lueker, G.S. (1974) *Structured Breadth First Search and Chordal Graphs*. Tech. Rep. TR-158, Princeton: Princeton University.
- Lukes, J.A. (1974) Efficient algorithm for the partitioning of trees. *IBM J. Res. Develop.*, 18, 217-224.
- Lukes, J.A. (1975) Combinatorial solution to the partitioning of general graphs, *IBM J. Res. Devel.*, 170-180.
- Machtey, M., Young, P. (1978) *An Introduction to the General Theory of Algorithms*. Nova York: North-Holland.
- Maculan Filho, N., Campello, R.E. (1994) *Algoritmos e Heurísticas: Desenvolvimento e Avaliação de Performance*. Niterói: Editora da UFF.
- Malhotra, V.M., Pramodh Kumar, M., Maheshwari, S.N. (1978) An  $O(|V|^3)$  algorithm for finding maximum flow in networks. *Inf. Proc. Letters*, 7, 277-278.
- Matula, D.W. (1968) A min-max theorem for graphs with application to graph coloring. *SIAM Review*, 10, 481-482.
- Matula, D.W., Marble, G., Isaacson, J.D. (1972) Graph Coloring Algorithms. In: Read, R.C. (ed.) *Graph Theory and Computing*. Nova York: Academic Press, 109-122.
- Mckee, T.A., McMorris, F.R. (1999) *Topics in Intersection Graphs*. SIAM.
- Menger, K. (1927) Zur allgemeine Kurventheorie. *Fund. Math.*, 10, 96-115.
- Méxas, M.P. (1982) *Um Estudo sobre Técnicas de Busca em Grafos e suas Aplicações*. Tese de Mestrado. COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

- Micali, S., Vazirani, V.V. (1980) An  $O(\sqrt{|V||E|})$  Algorithm for Finding Maximum Matching in General Graphs. *Proc. 21st Ann. Symp. on the Foundations of Computer Science, IEEE*, 17-27.
- Mithchen, J. (1976) On various algorithms for estimating the chromatic number of a graph. *The Computer Journal*, 19, 182-183.
- Moore, C., Mertens, S. (2011) *The Nature of Computation*. Oxford: Oxford University Press.
- Munkres, J. (1957) Algorithms for the assignment and transportation problems. *SIAM Journal*, 32-38.
- Mycielski, J. (1955) Sur le coloriage des graphs. *Colloq. Math.*, 3, 161-162.
- Nesetril, J., Milkova, E., Nesetrilova, H. (2001) Otakar Boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233, 3-36.
- Nijenhuis, A., Wilf, H.S. (1975) *Combinatorial Algorithms*. Nova York: Academic Press.
- Oliveira, A.A.F., Gonzaga, C.C. (1983) *Convergence Bounds for "Capacity" Max-Flow Algorithm*. Tech. Rep. Institut für Ökonometrie und Operations Research, Universität Bonn, Bonn.
- Ore, O. (1962) *Theory of Graphs*. American Math. Soc., Providence.
- Ore, O. (1967) *The Four-Color Problem*. Nova York: Academic Press.
- Pacitti, T., Atkinson, C.P. (1975) *Programação e Métodos Computacionais*. Vols. 1 e 2. Rio de Janeiro: LTC.
- Page, E.S., Wilson, L.B. (1973) *Information Representation and Manipulation in a Computer*. Londres: Cambridge University Press.
- Page, E.S., Wilson, L.B. (1979) *An Introduction to Computational Combinatorics*. Cambridge: Cambridge University Press.
- Papadimitriou, C.H. (1994) *Computational Complexity*. Reading: Addison-Wesley.
- Papadimitriou, C.H., Steiglitz, K. (1982) *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs: Prentice Hall.
- Petersen, J. (1891), Di theorie der regularen graphen. *Acta Math.*, 15, 193-220.
- Pfaltz, J.C. (1977) *Computer Data Structures*. Nova York: McGraw-Hill.
- Pombo, H.C.R. (1979) *Representação de Grafos em Computador*. Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro.
- Pratt, V. (1975) Every prime has a succinct certificate. *SIAM J. Comp.*, 4, 214-220.
- Prim, R.C. (1957) Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36, 1389-1401.
- Quinlan, J.R. (1986) Introduction of decision trees. *Machine Learning*, 1, 81-106.
- Rabin, M.O. (1976) *Probabilistic Algorithms and Complexity: New Directions and Recent Results*. Tramb, J.F. (ed.) Nova York: Academic Press, 21-40.
- Read, R.C., Tarjan, R.E. (1975) Bounds on backtrack algorithms for listing cycles, paths and spanning trees. *Networks*, 5, 237-252.
- Redei, L. (1934) Ein kombinatorischer Satz. *Acta Litt. Szeged*, 7, 39-43.
- Reingold, E.M., Nievergelt, J., Deo, N. (1977) *Combinatorial Algorithms: Theory and Practice*. Nova York: Englewood Cliffs.
- Rose, D.J., Tarjan, R.E. (1975) Algorithmic Aspects of Vertex Elimination. *Proc. 7<sup>th</sup> Ann. ACM Symp. Theory Comput.*, 245-254.
- Rose, D.J., Tarjan, R.E., Lueker, G.S. (1976) Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comp.*, 5, 266-283.
- Saaty, T., Kainen, P. (1977) *The Four Color Problem: Assaults and Conquests*. Nova York: McGraw-Hill.
- Santos, R.N. (1979) *Obtenção do Número Cromático de um Grafo*. Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro.
- Santos, V.F., Souza, U.S. (2015) *Uma Introdução à Complexidade Parametrizada*. Anais da 34<sup>a</sup> Jornada de Atualização em Informática, Congresso da Sociedade Brasileira de Computação, 232-273.
- Savulecu, S.C. (1980) *Grafos, Digrafos e Redes Elétricas*. Instituto Brasileiro Inst. Bras. Ed. Cient., São Paulo.
- Sethi, R. (1973) Complete register allocation problems. *SIAM J. Comp.*, 4, 226-248.
- Schrader, R. (1981) *A Note on Approximation Algorithms for Graph Partitioning Problems*. Rep. no. 81187-OR, Institut für Ökonometrie und Operations Research, Universität Bonn, Bonn.
- Schrijver, A. (2002) On the history of the transportation and the maximum flow problems. *Journal on Computer and System Sciences*, 61, 185-225.

- Schrijver, A. (2003) *Combinatorial Optimization*. Berlim: Springer-Verlag.
- Shier, D.R. (2004) Matchings and Assignments. In: Gross, J., Yellen, J. (eds.) *Handbook of Graph Theory*. CRC Press, 1103-1116.
- Shiloach, Y. (1978) *An O(nm log<sup>2</sup> n) Maximum Flow Algorithm*. Tech. Rep. ST AN-CS-78?802, Comp. Se. Dept., Stanford University, Stanford.
- Simon, I. (1981) *Linguagens Formais e Autômatos*. 2<sup>a</sup> Escola de Computação, Instituto de Matemática, Estatística e Ciência da Computação, Universidade de Campinas, Campinas.
- Simon, I. (1979) *Introdução à Teoria de Complexidade de Algoritmos*. 1<sup>a</sup> Escola de Computação, Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.
- Simões Pereira, J.M.S. (2009) *Matemática Discreta: Grafos, Redes e Aplicações*. Coimbra: Editora Luz da Vida.
- Sleator, D.D.K. (1980) *An O(nm log n) Algorithm for Maximum Network Flow*. Tese de doutorado, Comp. Se. Dept., Stanford University, Stanford.
- Spira, P.M. (1973) A new algorithm for finding all shortest paths in a graph of positive arcs in average time  $O(n^2 \log^2 n)$ , *SIAM J. Comp.*, 2, 28-32.
- Spinrad, J.P. (2003) *Efficient Graph Representations*. In: Fields Institute Monographs, vol. 19, American Mathematical Society.
- Standish, T.A. (1980) *Data Structure Techniques*. Reading: Addison-Wesley.
- Szwarcfiter, J.L. (1980) *On a Class of Acyclic Directed Graphs*. Mem. UCB/E RL M80/6, Elec. Res. Lab., University of California, Berkeley.
- Szwarcfiter, J.L. (1983) *Grafos e Algoritmos Computacionais*. Rio de Janeiro: Elsevier.
- Szwarcfiter, J.L., Markenzon, L. (2010) *Estruturas de Dados e Seus Algoritmos*. 3<sup>a</sup> ed. Rio de Janeiro: LTC.
- Szwarcfiter, J.L., Persiano, R.C.M. e Oliveira, A.A.F. (1985) Orientations with single source and sink. *Discrete Applied Mathematics*, 10, 313-321.
- Tarjan, R.E. (1972) Depth-first search and linear graph algorithms. *SIAM J. Comp.*, 1, 146-160.
- Tarjan, R.E. (1973) Enumeration of the elementary circuits of a directed graph. *SIAM J. Comp.*, 2, 211-216.
- Tarjan, R.E. (1974a) Finding dominators in directed graphs. *SIAM J. Comp.*, 3, 62-89.
- Tarjan, R.E. (1974b) Testing flow graph reducibility. *J. Comput. Sys. Sc.*, 9, 335-365.
- Tarjan, R.E. (1978) Complexity of combinatorial algorithms, *SIAM Review*, 20, 457-491.
- Tarjan, R.E. (1983) Data Structures and Network Algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics*, 44. Society for Industrial and Applied Mathematics.
- Tarry, G. (1895) Les problèmes des labyrinthes. *Nouvelles Ann. de Math.*, 14, 187.
- Terada, R. (1982) *Desenvolvimento de Algoritmos e Complexidade de Computação*. 3<sup>a</sup> Escola de Computação, Rio de Janeiro.
- Terada, R. (1991) *Introdução à Computação e à Estrutura de Dados*. McGraw Hill.
- Thorup, M. (1999) Undirected single-source shortest paths with positive integer weights in linear time. *J. of the ACM*, 46, 362-394.
- Turing, A. (1936) On computable numbers, with an application to the entscheidungs-problem. *Proc. London Math. Soc.*, Ser. 2, 42, 230-265.
- Tutte, W.T. (1966) *The Connectivity of Graphs*. Toronto: Toronto University Press.
- Valdes, J. (1978) *Parsing Flowcharts and Series Parallel Graphs*. Tech. Rep. STAN-CS-78-682, Comp. Sc. Dept., Stanford University, Stanford.
- Valdes, J., Tarjan, R.E., Lawler, E.L. (1979) The Recognition of Series Parallel Digraphs. *Proc. 11<sup>th</sup> Ann. ACM Symp. on Theory of Comp.*, Atlanta.
- Vazirani, V.V. (1994) A theory of alternating paths and blossoms for proving correctness of the  $O(\sqrt{V}E)$  general graph matching algorithm. *Combinatorica* 14, 71-109.
- Vazirani, V.V. (2012) A Simplification of the MV Matching Algorithm and its Proof, *arXiv:1210.4594 [cs.DS]*.
- Vazirani, V.V. (2014) A Proof of the MV Matching Algorithm. *Manuscript*.
- Veloso, P.A.S. (1979) *Máquinas e Linguagens: uma Introdução à Teoria de Autômatos*. 1<sup>a</sup> Escola de Computação, Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.
- Veloso, P., Santos, C., Azevedo, P., Furtado, A. (1984) *Estruturas de Dados*. Rio de Janeiro: Elsevier.

- Wakabayashi, Y. (1977) *Sobre Grafos Hamiltonianos*. Tese de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.
- Warshall, S. (1962) A theorem on boolean matrices. *J. of the ACM*, 9, 11-12.
- Weide, B. (1977) A survey of analysis techniques for discrete algorithms. *ACM Computing Surveys*, 9, 291-313.
- Wells, M.B. (1971) *Elements of Combinatorial Computing*. Oxford: Pergamon Press.
- Welsh, D.J.A., Powell, M.B. (1967) An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10, 85-86.
- West, D. (2001) *Introduction to Graph Theory*. 2<sup>a</sup> ed. Prentice-Hall.
- Whitney, H. (1932) Congruent graphs and the connectivity of graphs. *Amer. J.Math.*, 54, 150-168.
- Wilson, R. J. (1972), *Introduction to Graph Theory*, Oliver and Boyd, Edinburgh, 1972.
- Wilson, R.J., Beineke, L.W. (1979) *Applications of Graph Theory*. Nova York: Academic Press.
- Wood, D. (1987) *Theory of Computation*. Nova York: John Wiley & Sons.
- Yao, A.C.C. (1975) An  $O(|E| \log \log |V|)$  algorithm for finding minimum spanning trees, *Info. Proc. Let.*, 4, 21-23.
- Yen, J.Y. (1971) Finding the k shortest loopless paths in a network. *Management Sci*, 17, 712-716.
- Ziviani, N. (1999) *Projeto de Algoritmos com Implementação em C e em Pascal*. 4<sup>a</sup> ed. São Paulo: Pioneira Informática.
- Zykov, A.A. (1949) On some properties of linear complexes. *Mat. Sbornik*, 24, 163-188 (Amer. Math. Soc. Translation N. 79, 1952).

Página deixada intencionalmente em branco

Página deixada intencionalmente em branco

---

# Posfácio

---

Há uma diversidade de temas que poderiam ser abordados em um posfácio, sejam comentários ou considerações finais acerca da obra que foi apresentada, ou até fatos ocorridos com o autor, relacionados com o trabalho. No caso presente, decidi dedicar este espaço aos que propiciaram, através de seu trabalho, a realização deste projeto.

O trabalho para a preparação do livro contou com a ajuda inestimável, principalmente, de três colegas: Professor Fabiano de Souza Oliveira, Professor Paulo Eustáquio Duarte Pinto, e Professora Lucila Maria de Souza Bento. Os dois primeiros se dedicaram, primordialmente, à implementação dos algoritmos, na linguagem Python. O projeto inicial não contemplava implementações. A sua inclusão foi sugestão de Fabiano e Paulo, os quais realizaram todo o trabalho envolvido nos programas. Além disso, se dedicaram também à depuração e análise dos algoritmos, em si, bem como de suas provas de correção. Finalmente, contribuíram com sugestões de exercícios, incorporados ao texto. A Professora Lucila Maria de Souza Bento se dedicou ao trabalho de preparação gráfica do texto, em Latex. Essa tarefa incluiu a elaboração das figuras, bem como o projeto de paginação de todo o livro. Na realidade, o trabalho da Lucila excedeu o aspecto gráfico. Ela se dedicou também à correção de diversas partes do texto. Sou imensamente grato a esses colegas, sem os quais o livro com o atual formato e conteúdo não seria realizado.

Devo agradecer à Professora Celina Miraglia Herrera de Figueiredo, minha colega e professora titular da COPPE/UFRJ, que prontamente aceitou a tarefa de escrever o prefácio, que certamente valorizou o livro como um todo. Além disso, Celina disponibilizou um texto que havíamos escrito em conjunto sobre emparelhamentos de grafos.

Menciono ainda que o marco inicial deste projeto foi uma visita realizada por representantes da Editora Elsevier, ao Professor Vinicius Fernandes dos Santos, na UFMG. Por sugestão dele, fui contatado pela Elsevier, com a proposta de preparar esta obra, a partir meu livro anterior, *Grafos e Algoritmos Computacionais*, publicado pela Editora Campus, em 1983. Agradeço ao Vinicius pela iniciativa.

Sou grato pelo apoio recebido por parte dos colegas da UFRJ e da UERJ. Foi especialmente relevante o trabalho de leitura e revisão do texto, realizado por colegas e alunos. Além dos nomes acima mencionados, destacaria a revisão detalhada realizada por Bruno Porto Masquio, aluno de mestrado da UERJ.

Agradeço à Universidade Federal do Rio de Janeiro, instituição na qual sou docente há cerca de 50 anos, que propiciou a minha formação e carreira. Agradeço também à Universidade do Estado do Rio de Janeiro, que atualmente me recebe como pesquisador visitante: esta obra se beneficiou imensamente do meu contato com seus docentes. Destaco ainda as Universidades de Buenos Aires e Nacional de La Plata (Argentina), pelo apoio e intercâmbio contínuo com seus pesquisadores ao longo de cerca de 30 anos.

Finalmente, agradeço ao apoio e trabalho da Editora Elsevier, durante o decorrer do projeto.

**Jayme Luiz Szwarcfiter**  
Janeiro 2018

---

# Índice Remissivo

---

- árvore, 24  
altura, 28, 29  
centro, 27  
de caminhos mínimos, 144  
de decisão, 58  
de espalhamento, 27  
de largura, 76  
de largura lexicográfica, 79  
de profundidade, 66, 68, 71  
direcionada enraizada, 39  
enraizada, 28  
enraizada ordenada, 29  
estritamente binária, 29  
estritamente m-ária, 29  
geradora, 27  
geradora mínima, 102  
irrestrita de profundidade, 87  
livre, 28  
m-ária, 29  
M-alternante, 177  
peso, 102  
raiz, 28
- Agrawal, M., 234  
Aho, A. V., 18, 234  
ALGOL, 4  
algoritmo  
de Bellman-Ford, 148  
de Dijkstra, 144  
de Floyd, 149  
eficiente, 203  
guloso, 101  
húngaro, 179  
pseudopolinomial, 226  
alteração estrutural, 112  
ancestral, 28  
próprio, 28  
Andrade, M. C. Q., 51  
Appel, K., 3, 17, 51
- Araújo, G. S., 18  
aresta  
adjacente, 19  
capacidade, 123  
conectividade, 139  
contrária, 126  
convergente, 38  
de árvore, 71, 72, 78  
de avanço, 71, 72  
de cruzamento, 71, 72  
de retorno, 71, 72  
direta, 126  
divergente, 38  
elo, 27  
excesso da, 182  
implícita por transitividade, 94  
irmão, 78  
laço, 20  
pai, 78  
paralela, 20  
peso, 102  
primo, 78  
subdivisão, 34  
tio, 78
- Arora, S., 234  
articulação, 30, 68  
Ascencio. A. F. G., 18  
Atkinson, C. P., 18  
Azeredo, P., 18
- Baase, S., 18, 234  
Barak, B., 234  
Barbosa, R. M., 51  
Barron. D. W., 62  
base da flor, 186  
Baumert, L. D., 99  
Beineke, L. W., 51  
Bellman, R. E., 121, 144, 148, 170  
Berge, C., 51, 173, 201

- Berztiss, A. T., 18  
 Biggs, N. L., 17  
 bloco, 31  
 Boaventura Netto, P. O., 51  
 Bollobás, B., 51  
 Bondy, J. A., 51  
 Boruvka, O., 120, 121  
 Brandstädt, A., 51  
 Brelaz, D., 62  
 Busacker, R. G., 51  
 busca  
     em grafos, 63  
     em largura, 76  
     em largura lexicográfica, 79  
     em profundidade, 66, 70  
     geral, 65  
     irrestrita, 86  
     irrestrita em profundidade, 87  
 cadeia, 140  
 caminho, 21  
     aumentante, 127  
     comprimento, 21  
     elementar, 21  
     euleriano, 21  
     hamiltoniano, 21  
     M-alternante, 173  
     M-aumentante, 173  
     mínimo, 143  
     prefixo, 156  
     simples, 21  
     sufixo, 156  
     trajeto, 21  
 Campello, R. E., 18  
 Capobianco, M., 51  
 Cardoso, D. M., 51  
 Carré, B., 51  
 Cartwright, D., 51  
 Carvalho, L. R., 18  
 Cayley, A., 3, 17, 51  
 Cerioli, M. R., 18  
 Chazelle, B., 121  
 Cheriton, D., 120  
 Cherkassky, B., 141  
 Christofides, N., 51  
 ciclo, 21  
     elementar, 21  
     euleriano, 21  
     fundamental, 28  
     hamiltoniano, 21, 35  
     simples, 21  
     triângulo, 21  
 cláusula, 207  
 classe NP, 210  
 classe P, 206  
 clique, 24, 208  
 cobertura  
     por ciclos, 140  
     por vértices, 180  
     por vértices ponderada, 181  
 Cobham, A., 234  
 coloração, 36  
     aproximada, 55  
     k-coloração, 36  
 complexidade  
     algoritmos, 6  
     de melhor caso, 8  
     de pior caso, 8  
 componente  
     biconexa, 30, 68, 70  
     conexa, 21  
     fortemente conexa, 38  
 condensação, 113  
 conectividade, 30  
     biconexo, 31  
     de arestas, 30  
     de vértices, 30  
 conjunto  
     fundamental de ciclos, 28  
     independente de vértices, 24  
     maximal, 21  
     minimal, 21  
     parcialmente ordenado, 39  
 Cook, S. A., 234  
 Cormen, T. H., 99  
 Corneil, D. G., 100, 121  
 corte, 125  
     capacidade, 125  
     de arestas, 30  
     de vértices, 30  
     mínimo, 125  
 De Morgan, 17  
 demarcador, 69  
 Deo, N., 18, 51, 234  
 descendente, 28  
     próprio, 28  
 desvio, 152  
     simples, 156  
 detecção de ciclos negativos, 159

- Diestel, R., 51  
digrafo, 38  
  acíclico, 39  
  desconexo, 38  
  fecho transitivo, 39  
  fortemente conexo, 38  
  redução transitiva, 39  
  série paralelo geral, 95  
  série paralelo minimal, 95  
  unilateralmente conexo, 38  
Dijkstra, E. W., 120, 144, 170  
Dilworth, R. P., 141  
Dinic, E. A., 141  
Dirac, G. A., 51, 100  
Diverio, T. A., 18  
Dreyfus, S. E., 170  
Duan, R., 201  
  
Edmonds, J., 129, 141, 188, 201, 234  
emparelhamento, 171  
  induzido, 198  
  máximo, 171  
  maximal, 171  
  perfeito, 171  
  peso, 171  
Eppstein, D., 170  
equações de Bellman, 144  
esquema de eliminação perfeita, 83  
estrutura de adjacências, 41  
Euler, L., 2, 17  
Even, S., 18, 51, 141, 201, 234  
expressão booleana, 207  
  
Fáry, I., 51  
faces, 32  
fecho transitivo, 39  
Feofiloff, P., 18, 51  
Fernandes, C. G., 18  
Figueiredo, C. M. H., 18, 100, 201  
fila, 11  
  dupla, 12  
  dupla de entrada restrita, 12  
  dupla de saída restrita, 12  
floresta, 24  
  de caminhos mínimos, 146  
  de profundidade, 72  
  geradora, 27  
  húngara, 178  
  M-alternante, 177  
Floyd, R. W., 99, 149, 170  
fluxo, 123  
  
ilegal, 123  
máximo, 125  
maximal, 125  
valor, 123  
folha, 24  
Fonseca, G. D., 18  
fonte, 38  
Ford, L. R., 141, 148, 170  
forma normal conjuntiva, 207  
Francis Guthrie, 3  
Fraysseix, H., 99  
Fritsch, E., 51  
Fritsch, R., 51  
frondes, 66  
Fulkerson, D. R., 100, 141  
Furtado, A. L., 18, 51  
  
Gabow H. N., 201  
Galil, Z., 141, 201  
Gallai, T., 51, 52  
Garey, M. R., 62, 234  
Gerards, A. M. H., 201  
Ghouila-Houri, A., 52  
Gilmore, P. C., 52  
Goldbach, E., 18  
Goldbach, M., 18  
Goldberg, A. V., 141  
Golomb, S. W., 99  
Golumbic, M. C., 51, 100  
Gondran, M., 51  
Gonzaga, C. C., 141  
Goodman, S. E., 18  
grafo, 19  
  acíclico, 21  
  bipartido, 22, 23  
  bipartido completo, 23  
  bloco-articulação, 48  
  centro, 26  
  complemento, 22, 23  
  completo, 22, 23  
  conexo, 21, 199  
  cordal, 82  
  de comparabilidade, 49  
  de Petersen, 48  
  desconexo, 21  
  direcionado, 38  
  euleriano, 21  
  hamiltoniano, 21, 35  
  imersível, 32  
  isomorfo, 20

- k-colorível, 36  
 k-crítico, 37  
 maximal planar, 49  
 não direcionado, 19  
 outerplanar, 49  
 planar, 32  
 ponderado, 181  
 regular, 20  
 representação geométrica, 19  
 representação planar, 32  
 rotulado, 24  
 subdivisão, 34
- Graham, B., 121  
 grau, 20  
   de entrada, 38  
   de saída, 38
- Greene, D. H., 18  
 Gross, O. A., 100  
 Guthrie, 17
- Habib, M., 100  
 Hadlock, F. O., 121  
 Haken, W., 3, 17, 51  
 Hall, M., 201  
 Hall, P., 141, 172, 201  
 Halldórsson, M. M., 62  
 Harary, F., 51  
 Harrison, M. C., 18  
 haste, 186  
 Heawood, P. J., 17  
 Hedetniemi, S. T., 18  
 Henzinger, M. R., 170  
 Hoffman, A. J., 52  
 Hopcroft, J. E., 17, 18, 99, 100, 201, 234  
 Horowitz, E., 18, 234  
 Hu, T. C., 18
- implementações em Python, 42  
 Isaacson, J. D., 62  
 Itai, A., 141
- Jensen, T. R., 62  
 Johnson, D. B., 100  
 Johnson, D. S., 62, 234  
 Jordan, C., 17, 51
- k-ésimos caminhos mínimos, 151  
   simples, 156
- König, 3, 17, 181, 182, 201  
 Königsberg, 17  
 Kahn, A. B., 62
- Kainen, P., 51  
 Kariv, O., 201  
 Karp, R. M., 129, 141, 201, 234  
 Karzanov, A. V., 141  
 Kase, R. H., 62  
 Kayal, K., 234  
 Kempe, A. B., 17  
 Kirchhoff, G., 3, 17, 51  
 Klein, P., 170  
 Kleinberg, J., 141, 170  
 Knuth, D. E., 18, 51, 62, 99, 234  
 Kowaltowski, T., 18  
 Kruskal Jr., J. B., 120  
 Kuhn, H. W., 201  
 Kuratowski, K., 3, 17, 51
- Lageweg, B. J., 234  
 Lamb, L. C., 18  
 Lasser, D. J., 62  
 Lawler, E. L., 18, 100, 121, 170, 201, 234  
 Le, V. B., 51  
 Leiserson, C. E., 99  
 Lemos, H., 18  
 Lenstra, J. K., 234  
 Levin, L. A., 234  
 Lewis, H. R., 17  
 lista, 10  
   circular duplamente encadeada, 11  
   de adjacências, 41  
   duplamente encadeada, 11  
   encadeada, 10  
   literais, 207  
 Lloyd, E. K., 17  
 Lovász, L., 51, 201  
 Lucas, E., 99  
 Lucchesi, C. L., 18, 51  
 Lucena, C. J. P., 18  
 Lueker, G. S., 100  
 Lukes, J. A., 121
- M-exposto, 171  
 M-flor, 186  
 M-floração, 186  
 M-saturado, 171  
 método húngaro, 176  
 Méxas, M. P., 100  
 Machtey, M., 18  
 Maculan Filho, N., 18  
 Maheshwari, S. N., 133, 141  
 Malhotra, V. M., 133, 141  
 Marble, G., 62

- Markenzon, L., 18  
matriz  
  booliana, 229  
  de adjacências, 40  
  de distâncias, 143  
  de incidências, 41  
Matula, D. W., 62  
McConnell, R., 100  
Mckee, T. A., 51  
McMorris, F. R., 51  
Meidanis, J., 100  
Mello, C. P., 100  
Menezes, P. B., 18  
Menger, K., 3, 17, 51  
Mertens, S., 234  
Micali, S., 141, 201  
Milkova, E., 121  
Minoux M., 51  
Mithchen, J., 62  
Miyazawa, F. K., 18  
Molluzzo, J. C., 51  
Moore, C., 234  
multigrafo, 20  
Munkres, J., 201  
Murty, U. S. R., 51  
Mycielski, J., 51  
  
número cromático, 36, 113  
Naamad, A., 141  
Nesetril, J., 121  
Nesetrilova, H., 121  
Nievergelt, J., 18, 234  
Nijenhuis, A., 18  
Norman, R. Z., 51  
notação  $\Omega$ , 8  
notação  $O$ , 8  
NP-completo, 203  
  NP-completo forte, 226  
  NP-completo fraco, 226  
  
Oliveira, A. A. F., 99, 141  
ordem de nível, 63  
ordenação  
  lexicográfica, 80  
  topológica, 57  
Ore, O., 51  
Ossona de Mendez, P., 99  
  
Pacitti, T., 18  
Page, E. S., 18  
Papadimitriou, C. H., 18, 201, 234  
  
particionamento de árvores, 106, 226  
passeio, 21  
  M-alternante, 186  
  M-aumentante, 186  
  M-aumentante minimal, 186, 189  
Paul, C., 100  
Persiano, R. C. M., 99  
Petersen, J., 51, 200, 201  
Pfaltz, J. C., 18  
pilha, 11  
planaridade, 34  
Plummer, M. D., 51, 201  
Pombo, H. C. R., 18  
ponte, 30  
ponte de Königsberg, 2  
Powell, M. B., 62  
Pramodh Kumar, M., 133, 141  
preordem, 63  
Prim, R. C., 120  
Prins, G., 51  
problema  
  2-satisfabilidade, 221  
  3-satisfabilidade, 221  
  caixeiro viajante, 205  
  caminho mínimo condicionado, 231  
  ciclo hamiltoniano direcionado, 208  
  ciclo hamiltoniano não direcionado, 209  
  clique, 208  
  clique máxima, 213  
  cobertura por vértices, 208  
  coloração, 209  
  complemento, 214  
  conjunto de arestas de realimentação, 209  
  conjunto de vértices de realimentação, 209  
  conjunto independente de vértices, 208  
  de decisão, 204  
  de localização, 204  
  de otimização, 204  
  do fluxo máximo, 123  
  extensão, 220  
  intratável, 203  
  isomorfismo de subgrafos, 210  
  nímeros compostos, 215  
  NP-completo, 203  
  numérico, 226  
  particionamento de árvores, 224  
  restrição, 220  
  satisfatibilidade, 207, 220  
  tratável, 203  
profundidade