

GRAFOS E ALGORITMOS COMPUTACIONAIS

Este texto corresponde a um curso introdutório de algoritmos e grafos, matéria atual, ainda em formação, e que constitui tema relevante do ponto de vista prático, tanto em computação, quanto em matemática, pois são inúmeros os problemas que podem ser resolvidos mediante a modelagem em grafos e posterior utilização de um programa de computador, implementando um algoritmo conveniente.

A preocupação em desenvolver processos que utilizem o computador de forma eficiente é, pois, característica básica que distingue a obra, tornando-a indispensável para os estudantes e profissionais da área, que a ela poderão recorrer também como fonte bibliográfica, graças ao grande número de referências listadas, especialmente em língua portuguesa.

• Jayme Luiz Szwarcfiter, o autor, é Ph.D. pela University of Newcastle Upon Tyne, e trabalha atualmente na Universidade Federal do Rio de Janeiro, como pesquisador do Núcleo de Computação Eletrônica e como professor da COPPE e do Instituto de Matemática.

JAYME LUIZ SZWARCFITER

GRAFOS E ALGORITMOS COMPUTACIONAIS



JAYME LUIZ SZWARCFITER

Professor do Núcleo de Computação Eletrônica,
do Instituto de Matemática e da COPPE da UFRJ.

GRAFOS E ALGORITMOS COMPUTACIONAIS

EDITORIA CAMPUS LTDA.
Rio de Janeiro

© 1984, Editora Campus Ltda.

Todos os direitos reservados e protegidos pela Lei 5988 de 14/12/1973.
Nenhuma parte deste livro poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Todo o esforço foi feito para fornecer a mais completa e adequada informação. Contudo a editora não assume responsabilidades pelo uso da mesma.

Capa
Otavio Studart

Diagramação, composição, paginação e revisão

Editora Campus Ltda.

Rua Japeri 35 Rio Comprido

Tel.: (021) 284 8443 PABX

tel.: (21) 264-8445 FAX
20261 Rio de Janeiro RJ Brasil

Endereço telegráfico: CAMPUSRIO

ISBN 85-7001-138-5

Ficha Catalográfica

CIP-Brasil. Catalogação-na-fonte
Sindicato Nacional dos Editores de Livros, RJ

S998g Szwarcfiter, Jayme Luiz.
Grafos e algoritmos computacionais / Jayme Luiz Szwarcfiter. —
Rio de Janeiro : Campus, 1984.

Bibliografia
ISBN 85-7001-138-5

I. Título

CDD - 511.5
511.8
001.424
CDU - 517
681 3:51

83-0611

*Para Cláudio
e
Lila*

PREFÁCIO

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 possível posterior implementação em computador. Nesse sentido, a eficiência de tempo e espaço de um processo são fatores básicos para a 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.

A apresentação se desenvolve principalmente em torno da descrição de técnicas gerais, utilizadas na elaboração de algoritmos em grafos. 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 — método 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 em capítulos anteriores. Além disso, é uma ilustração didática do estudo de complexidade. O último capítulo des-

creve 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 do NP-completo em algoritmos para grafos torna quase obrigatória a sua inclusão em um livro desta natureza.

A bibliografia referenciada, em geral, é mais abrangente do que o texto. Contudo, este fato é proveitoso para leitores que desejem 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.

Este livro foi baseado em um texto do autor, escrito para o curso Algoritmos e Grafos da 3ª Escola de Computação, realizado na Pontifícia Universidade Católica, 1982. Gostaria de agradecer à Comissão Organizadora deste evento pela oportunidade concedida que deu origem ao presente trabalho.

Para a tarefa de depuração de erros obtive valioso auxílio dos meus alunos da COPPE, Universidade Federal do Rio de Janeiro (disciplina Teoria Computacional de Grafos, 1981 e 82) e do Instituto de Matemática e Estatística, Universidade de São Paulo (disciplina Algoritmos e Grafos, 1982).

Várias pessoas contribuíram para a realização desse trabalho, através de críticas, sugestões ou apoio. Em especial, gostaria de manifestar meus agradecimentos a Antonio Alberto F. Oliveira, Cesar José dos Santos, Cláudio Leonardo Lucchesi, Edgar Dias Batista Jr., Imre Simon, José Fábio M. Araújo, Luis Antonio Carneiro C. Couceiro, Maria E. del Pilar M. Gonçalves e Paulo Augusto Veloso.

Jayme Luiz Szwarcfiter
abril, 1983

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

1.1	Considerações Iniciais	17
1.2	Apresentação dos Algoritmos	20
1.3	Complexidade de Algoritmos	23
1.4	Estrutura de Dados	29
1.5	Exercícios	33
1.6	Notas Bibliográficas	33

CAPÍTULO 2 – UMA INICIAÇÃO À TEORIA DOS GRAFOS

2.1	Introdução	35
2.2	Os Primeiros Conceitos	35
2.3	Árvores	43
2.4	Conectividade	52
2.5	Planaridade	56
2.6	Ciclos Hamiltonianos	60
2.7	Coloração	62
2.8	Grafos Direcionados	64
2.9	Representação de Grafos	68
2.10	Exercícios	71
2.11	Notas Bibliográficas	73

CAPÍTULO 3 – TÉCNICAS BÁSICAS

3.1	Introdução	75
3.2	Processo de Representação	75

3.3	Adjacência	76
3.4	Ordenação de Vértices ou Arestas	76
3.5	Coloração Aproximada	77
3.6	Ordenação Topológica	79
3.7	Recursão	81
3.8	Árvores de Decisão	81
3.9	Limite Inferior para Ordenação	83
3.10	Exercícios	83
3.11	Notas Bibliográficas	84

CAPÍTULO 4 – BUSCA EM GRAFOS

4.1	Introdução	85
4.2	Algoritmo Básico	86
4.3	Busca em Profundidade	88
4.4	Biconectividade	91
4.5	Busca em Profundidade – Dígrafos	95
4.6	Componentes Fortemente Conexos	99
4.7	Busca em Largura	103
4.8	Busca em Largura Lexicográfica	105
4.9	Reconhecimento de Grafos Cordais	110
4.10	Busca Irrestrita	116
4.11	Exercícios	120
4.12	Notas Bibliográficas	124

CAPÍTULO 5 – OUTRAS TÉCNICAS

5.1	Introdução	125
5.2	Algoritmo Guloso	125
5.3	Árvore Geradora Máxima	126
5.4	Programação Dinâmica	130
5.5	Particionamento de Árvores	132
5.6	Alteração Estrutural	142
5.7	Número Cromático	143
5.8	Exercícios	146
5.9	Notas Bibliográficas	148

CAPÍTULO 6 – FLUXO MÁXIMO EM REDES

6.1	Introdução	149
6.2	O Problema do Fluxo Máximo	149

6.3	O Teorema do Fluxo Máximo – Corte Mínimo	152
6.4	Um Primeiro Algoritmo	156
6.5	Um Algoritmo O ($n m^2$)	157
6.6	Um Algoritmo O ($n^2 m$)	159
6.7	Um Algoritmo O (n^3)	161
6.8	Exercícios	165
6.9	Notas Bibliográficas	166

CAPÍTULO 7 – PROBLEMAS NP-COMPLETO

7.1	Introdução	169
7.2	Problemas de Decisão	170
7.3	A Classe P	173
7.4	Alguns Problemas Aparentemente Difíceis	174
7.5	A Classe NP	179
7.6	A Questão P = NP	183
7.7	Complementos de Problemas	184
7.8	Transformações Polinomiais	186
7.9	Alguns Problemas NP-Completo	189
7.10	Restrições e Extensões de Problemas	192
7.11	Algoritmos Pseudopolinomiais	194
7.12	Exercícios	196
7.13	Notas Bibliográficas	197

REFERÊNCIAS	199
-----------------------	-----

ÍNDICE DE ALGORITMOS	209
--------------------------------	-----

ÍNDICE ALFABÉTICO	211
-----------------------------	-----

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 entre os conjuntos S e R .
$S \times R$	produto cartesiano dos conjuntos S e R .
$ S $	cardinalidade do conjunto S .

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 .

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

$G(V, E)$	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 .
$G + S$	grafo obtido de G pela inclusão do conjunto de vértices (arestas) S .
\bar{G}	complemento do grafo G .
K_n	grafo completo com n vértices.
$K_{n,m}$	grafo bipartite $(V_1 \cup V_2, E)$ 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 .
$X(G)$	número cromático do grafo G .

CAPÍTULO 1

INTRODUÇÃO

1.1 – Considerações Iniciais

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 somente 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.

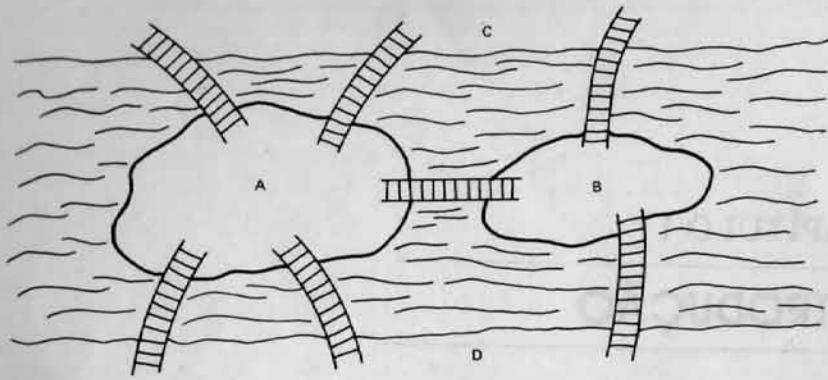


Figura 1.1. O problema da ponte de Königsberg

A solução do problema acima 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 viriam contribuir enormemente para despertar o interesse pela área. O primeiro desses fatos 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.

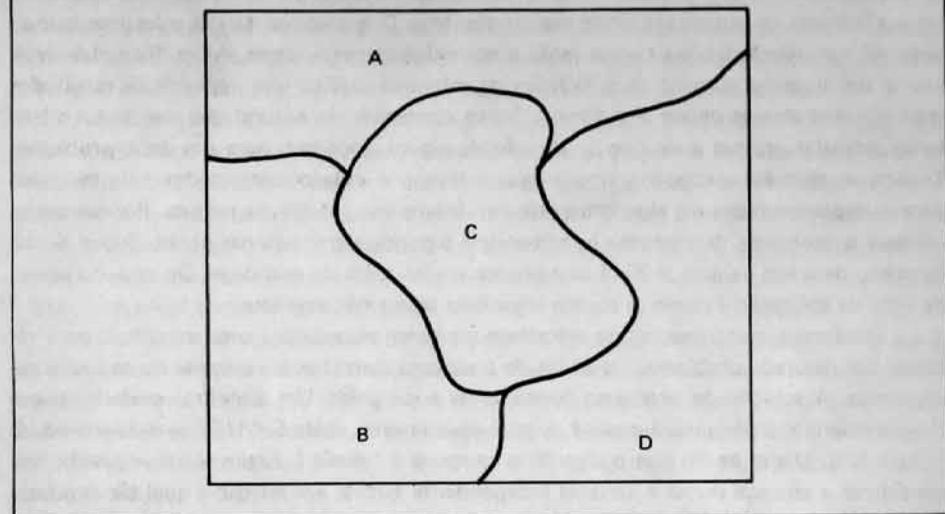


Figura 1.2. Quatro cores são necessárias

No problema do caminho hamiltoniano existem n cidades. Cada par de cidades pode ser adjacente 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, Menger, entre outros. Os anos mais recentes confirmam, de certa forma, a idéia 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 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 corrente, através da formalização da noção de uma 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.

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, acima descrito, 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.

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 a qual são produzidas as saídas do algoritmo, bem como são analisados os comportamentos de tempo e espaço do mesmo.

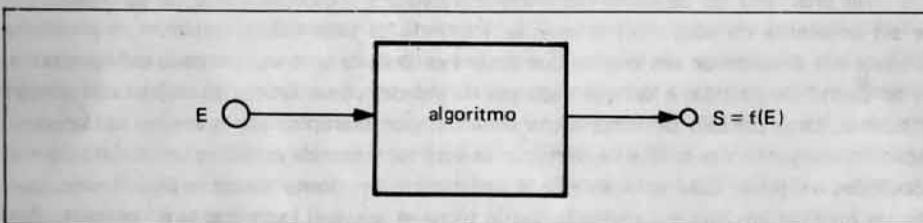


Figura 1.3. Um algoritmo

1.2 – Apresentação dos Algoritmos

Os algoritmos neste texto serão apresentados em uma linguagem quase livre de formato, contendo porém sentenças especiais com significado predefinido. 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 freqüência 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. Possui uma estrutura cujo objetivo seria o de facilitar a descrição e implementação dos processos. Esta estrutura deve ser sempre obedecida na apresentação, mesmo quando se utiliza o português livre de formato. As idéias 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 $t, t \neq r$, 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 necessariamente contíguas. Além disso, um bloco é sempre formado por uma seqüência 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} , $1 \leq j < k$.

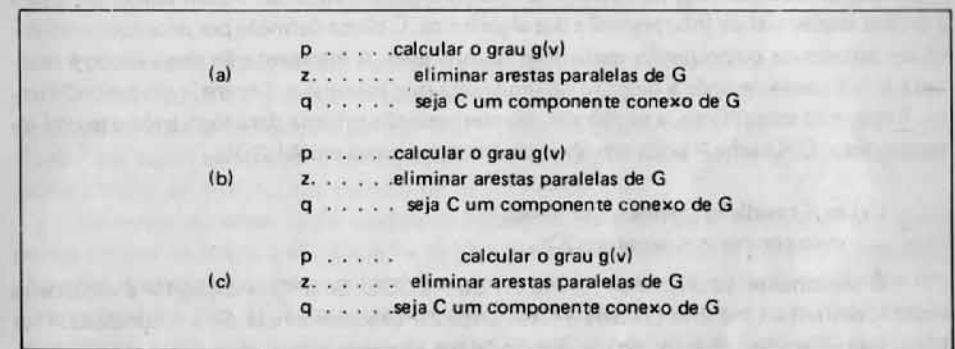


Figura 1.4. Linhas de um algoritmo

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 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 *bloco definido* por r , que é o maior de todos que se iniciam na linha r .

Além da estrutura acima descrita, 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. Em geral, 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 seqüencial de interpretação dos algoritmos. O bloco definido por *procedimento* deve ser saltado na computação seqüencial do processo. A interpretação desse bloco é realizada imediatamente após a deteção de uma linha que invoque o <nome> do procedimento. Encerrado este último, a seqüência 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 de 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.

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 seqüência $s(1), s(2), \dots, s(n)$ de termos. O objetivo é inverter a sua ordem, isto é, rearranjar os termos na seqüência, de modo que $s(n)$ apareça na posição 1, $s(n-1)$ na posição 2, e assim por diante. O algoritmo 1.1 seguinte descreve o processo.

algoritmo 1.1: Inversão de uma seqüência

dados seqüência s_1, s_2, \dots, s_n
para $j = 1, 2, \dots, \lfloor n/2 \rfloor$ efetuar
 $b := s(j)$
 $s(j) := s(n - j + 1)$
 $s(n - j + 1) := b$

1.3 – 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 no objetivo de procurar elaborar algoritmos que sejam eficientes, o mais possível. 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 da década de 60. Se bem que as informações obtidas através desse processo sejam também úteis, os critérios mencionados não permitem aferir, realmente, 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 do 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 que seja analítico, para avaliação da eficiência. O critério descrito a seguir tenta se aproximar deste objetivo.

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 acima mencionado, 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* (fig. 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 de 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 , ..., 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.

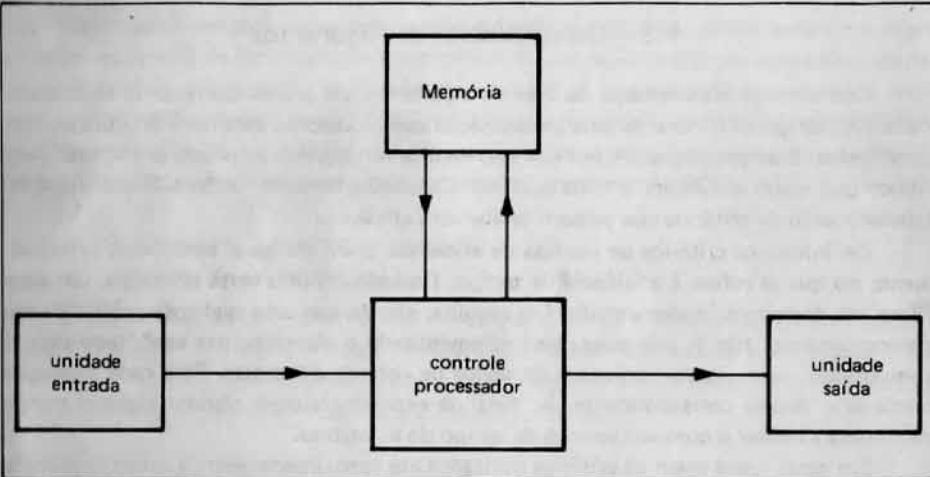


Figura 1.5. Um modelo computacional

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 casos 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 apenas 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* α à computação de uma instrução do programa P que o implementa. A *complexidade local* do algoritmo α 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 α 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 freqüentemente. 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. Conseqüentemente, 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, 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 freqüentes. 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 $c_1 x^2 + c_2 x$ 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 complexidades assintóticas do algoritmo, para entradas suficientemente grandes. Isto é, a complexidade de melhor caso corresponde a um limite superior 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 Ω 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, $5n^2 + 2n + 3 \in \Omega(n^2)$. Também $2n^3 + 5n \in \Omega(n^2)$, 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 α que resolva P requerer pelo menos $O(g)$ passos. Isto é, se $O(f)$ for a complexidade (pior caso) de α então g é $O(f)$. Por exemplo, se $\Omega(n^2)$ for um limite inferior para P não poderá existir algoritmo que o resolva em $O(n)$ passos. Um algoritmo α_0 que possua complexidade $O(g)$ é denominado *ótimo* e, nesse caso, g é o *limite inferior máximo* de P . Observe que α_0 é o algoritmo de complexidade mais baixa dentre todos os que resolvem P . Este novo indicador é obviamente importante e, freqüentemente, 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 1.2 para inversão da ordem dos termos de uma seqüência. A sua entrada consiste da seqüência s_1, \dots, s_n , onde cada s_i possui tamanho constante. Ou seja, a entrada possui tamanho $O(n)$. O procedimento corres-

ponde 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 seqüência 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 seqüência 2 5 3 4 apresenta duas inversões 5 3 e 5 4. A seqüência 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 seqüência

dados seqüência $S = s_1, \dots, s_n$
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 seqüência 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, 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 seqüência 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.

No algoritmo 1.2, a escolha da inversão a ser considerada é arbitrária. Para tornar o processo mais eficiente pode-se adotar um critério segundo o qual as inversões consideradas são sempre entre elementos consecutivos na sequência. Isto é, no passo inicial, o problema é restrito à subseqüência $S(1) = s_1$, já ordenada trivialmente. No passo geral, supõe-se que esteja ordenada a subseqüência $S(j - 1)$ formada pelos $j - 1$ elementos iniciais de S . Acrescente s_j à direita em $S(j - 1)$. A idéia 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 de S se encontrar na primeira posição da subseqüência, então $S(j)$ estará também ordenada. A formulação seguinte descreve o processo.

algoritmo 1.3: Ordenação de uma seqüência

```

dados seqüência S = s1, ..., sn
para j = 1, ..., n - 1 efetuar
    k := j
    enquanto sk > sk+1 e k ≥ 1 efetuar
        trocar de posição sk com sk+1
        k := k - 1

```

O efeito principal produzido por esta alteração é tornar constante o número de operações necessárias para identificar uma inversão. Assim sendo, a complexidade da ordenação decresce para $O(n + l)$. Isto é, as complexidades de pior e melhor caso do algoritmo acima são $O(n^2)$ e $O(n)$, respectivamente.

É possível melhorar uma vez mais a eficiência do processo de ordenação. No algoritmo 1.3, cada novo elemento s_j de S é deslocado, sucessivamente, uma posição para a esquerda na subseqüência $S(j - 1)$, até desaparecerem as inversões existentes. Como alternativa, é possível determinar, de maneira mais breve, a posição p em que s_j deve ser inserido em $S(j - 1)$ para formar a subseqüência ordenada $S(j)$. O objetivo é evitar que seja sempre percorrido todo o percurso, desde a posição mais à direita em $S(j - 1)$ até p . Isto é, o efeito de cada comparação de s_j com um elemento de $S(j - 1)$ pode produzir a eliminação de uma única inversão. Em contrapartida, o algoritmo 1.4 compara s_j com o elemento central s_c de $S(j - 1)$. Em consequência, podem ser eliminadas até $\lceil |S(j - 1)|/2 \rceil$ inversões com uma só comparação. Esta técnica é conhecida pelo nome de *busca binária*.

algoritmo 1.4: Ordenação de uma seqüência

```

dados seqüência S = s1, ..., sn
para j = 2, ..., n efetuar
    a := 1
    b := j - 1
    c := ⌊(a + b)/2⌋
    enquanto a ≠ b efetuar
        se sj ≤ sc então b := c caso contrário a := c
        c := ⌊(a + b)/2⌋
    se sj ≤ sc inserir sj à esquerda de sc em S
    caso contrário inserir sj à direita de sc em S

```

Para obter a complexidade do processo acima, observe que a inserção de s_j em qualquer posição de S pode ser realizada em tempo constante, caso sejam utilizadas estruturas de dados convenientes (ver seção 1.4). Portanto, a complexidade do algoritmo é determinada pelo número de passos efetuados no bloco definido pela sentença:

enquanto $a \neq b$ efetuar.

Para avaliar este número, note que o termo $b - a + 1$ corresponde ao comprimento da subseqüência em consideração. A cada iteração do bloco acima, este comprimento se reduz à metade, até atingir o valor 1. Para cada j , $2 \leq j \leq n$, o comprimento inicial da subseqüência é $j - 1$. Consequentemente, o número de passos efetuados no bloco em discussão é $O(\log j)$. Considerando $j = O(n)$ e levando em conta a variação de j , conclui-se que a complexidade do algoritmo é $O(n \log n)$. É possível provar (seção 3.9) que o algoritmo 1.4 é ótimo. Ou seja, qualquer algoritmo de ordenação que opere por comparações efetua $\Omega(n \log n)$ passos no pior caso.

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. Freqüentemente, estas podem ser obtidas sem maiores dificuldades através de expressões na notação O . Por exemplo, os algoritmos de ordenação acima descritos possuem complexidades de espaço de pior e melhor caso, ambas iguais a $O(n)$.

1.4 – 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 seqüência 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 seqüencialmente 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 seqüencial* de L , enquanto a lista L recebe o nome de *lista seqüencial* ou *vetor*.

Um outro modo de representar L consiste em alocar seus 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} .

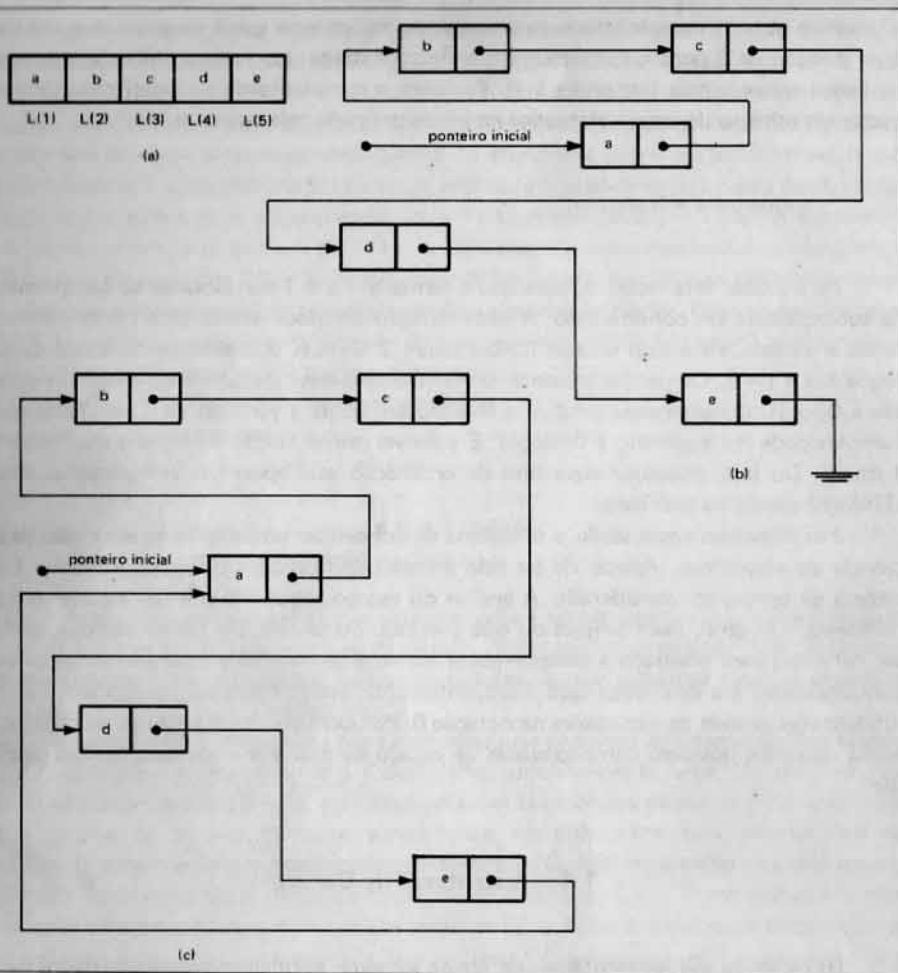


Figura 1.6. Representações diferentes de uma lista

Define-se também $t_n = \phi$, 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 $t_0 = \phi$.

Uma lista $L = \{a_1, \dots, a_n\}$, na representação por ponteiros e tal que o ponteiro t_n informa a localização do elemento a_1 (ao invés de $t_n = \phi$), recebe o nome de *lista circular*. As figuras 1.6(a), (b) e (c) ilustram uma lista $L = \{a, b, c, d, e\}$, nos casos, respectivamente,

mente, em que L é um vetor, uma lista encadeada e uma lista circular. A notação gráfica \rightarrow | | corresponde ao símbolo ϕ .

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 $s_1 = t_n = \phi$. Da 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*.

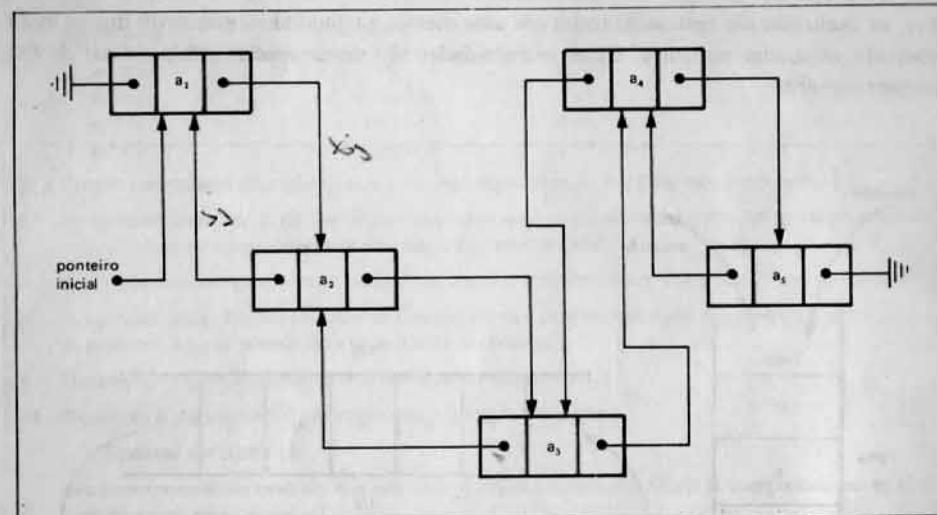


Figura 1.7. Uma lista duplamente encadeada

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 freqüentes. Uma *pilha* é uma lista em que todo elemento a ser excluído é necessariamente o último incluído. A idéia 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 seqüencial. Assim sendo, quando L é não-vazia é sempre possível identificar o primeiro e o último elementos da seqüência, os quais são denominados *extremidades* de L . Em particular, as operações de inclusão e exclusão nas listas especiais acima 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 *topo 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.

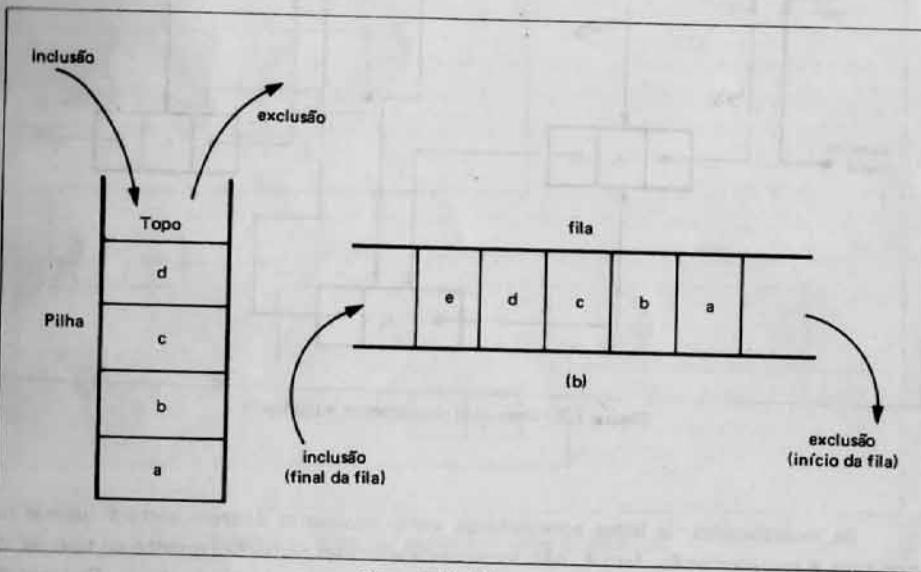


Figura 1.8. Pilha e fila

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 o 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.

1.5 – EXERCÍCIOS

- | <i>algoritmo A</i> | <i>algoritmo B</i> | <i>algoritmo C</i> |
|--------------------|--------------------|--------------------|
| $a := b$ | $a := b$ | $a := b$ |
| $c := d$ | $c := d$ | $c := d$ |
| $e := f$ | $e := f$ | $e := f$ |
| $g := h$ | $g := h$ | $g := h$ |
- 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 1.2, como proceder em caso de linha continuação, de modo a não contrariar as regras relativas à estrutura da linguagem?
- 1.3 Suponha um algoritmo α com m linhas e b blocos disjuntos cuja união contenha todas as linhas. Supondo m fixo, determinar a estrutura de α para que (i) b seja mínimo, (ii) máximo.
- 1.4 Determinar o menor número de blocos em que cada um dos algoritmos abaixo pode ser dividido.
- 1.5 Qual é o bloco definido pela linha $e := f$, nos algoritmos A, B e C do exercício acima?
- 1.6 A complexidade local de um algoritmo, para uma certa entrada, pode ser assintoticamente maior do que a complexidade de pior caso. Certo ou errado?
- 1.7 Definir as complexidades local, assintótica, de pior e melhor caso, correspondentes ao espaço.
- 1.8 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?
- 1.9 Determinar a complexidade de melhor caso do algoritmo 1.4.
- 1.10 Modificar o algoritmo 1.4, de modo que a iteração do bloco
enquanto $a \neq b$ efetuar
seja interrompida no caso em que um valor $s_j = s_c$ é encontrado. Quais as complexidades de pior e melhor caso dessa variante?
- 1.11 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?
- 1.12 Diz-se que uma pilha S *realiza* a permutação p de $1, \dots, n$ quando existe uma seqüência 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 $1, \dots, n$.
- 1.13 Resolver o problema acima, 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.
- 1.14 Uma k -pilha é uma lista em que todo elemento excluído é algum dentre os k elementos incluídos mais recentemente. Resolver o problema 1.12, com uma k -pilha substituindo a pilha. Supor, inicialmente, $k = 2$.

1.6 – 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 inclui uso intenso de computadores. Os desenvolvimentos importantes na teoria de grafos, produzidos por volta da década de 30 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 autônomas. Machtey e Young (1978) trata da teoria geral de algoritmos. Hopcroft e Ullman (1979) abrange ambos os temas. A literatura brasileira, nessa área, compreende Carvalho (1981), Lucchesi e outros (1979), Imre Simon (1981), Istvan Simon (1979) e Veloso (1979). O estudo de técnicas de algoritmos e suas complexidades é encontrado em Aho, Hopcroft e Ullman (1974), Baase (1978), Goodman e Hedetniemi (1977) e Horowitz e Sahni (1978). Este estudo constitui também o tema central 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 (1968). Mencionam-se ainda Berztiss (1971), Harrison (1973), Horowitz e Sahni (1976), Page e Wilson (1973), Pfaltz (1977), Standish (1980), entre outros. Os textos brasileiros em estrutura de dados são Furtado e outros (1982) e Lucena (1972). Um trabalho específico sobre representação de grafos é Pombo (1979), em português. Algoritmos em combinatoria 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). Os exercícios 1.10 e 1.12 são baseados em Knuth (1968).

CAPÍTULO 2

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 . G é chamado *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 $G_1(V_1,$

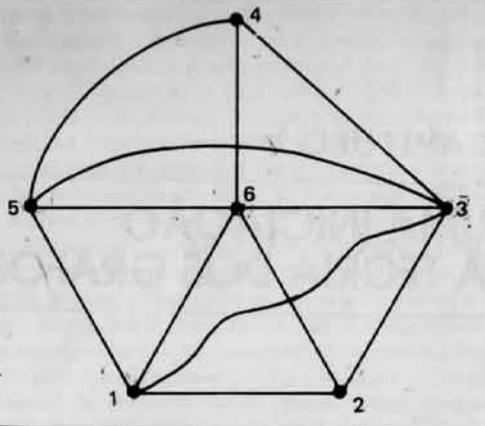


Figura 2.1. Um grafo (V, E) e uma representação geométrica do mesmo

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

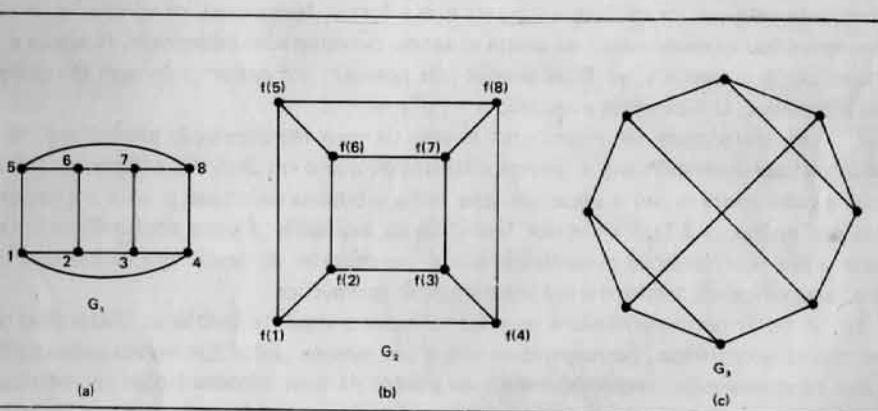


Figura 2.2. Grafos isomorfos e não isomorfos

Algumas vezes é útil se 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* (fig. 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* (fig. 2.3(b)).

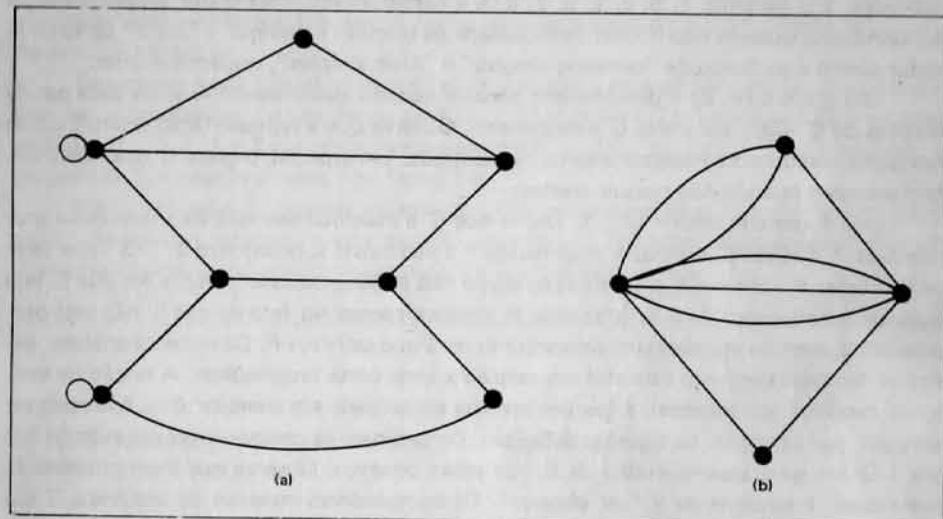


Figura 2.3. Um grafo com laços e um multigrafo

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 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 seqüência de vértices v_1, \dots, v_k tal que $(v_j, v_{j+1}) \in E$, $1 \leq j < k - 1$, é denominado *caminho* de v_1 a 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 seqüência recebe o nome de *caminho simples* ou *elementar*. Se as arestas forem distintas, a seqüência denomina-se *trajeto*. Por exemplo, no grafo da figura 2.2(a), a seqüência 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 con-

siderados idênticos se um deles puder ser obtido do outro, através de uma rotação 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 ambigüidade 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 $S'' \supset S'$, que também satisfaz P . Observe que a definição acima não implica necessariamente em que S' seja o *maior* subconjunto de S satisfazendo P . Implica apenas no 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 freqüentemente encontrada em combinatória. Ela pode ser aplicada, por exemplo, na seguinte definição. Denominam-se *componentes conexos* de um grafo G aos subgrafos maximais de G que sejam conexos. Observe que a propriedade P , nesse caso, é equivalente a “ser conexo”. Os componentes conexos 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 conexos.

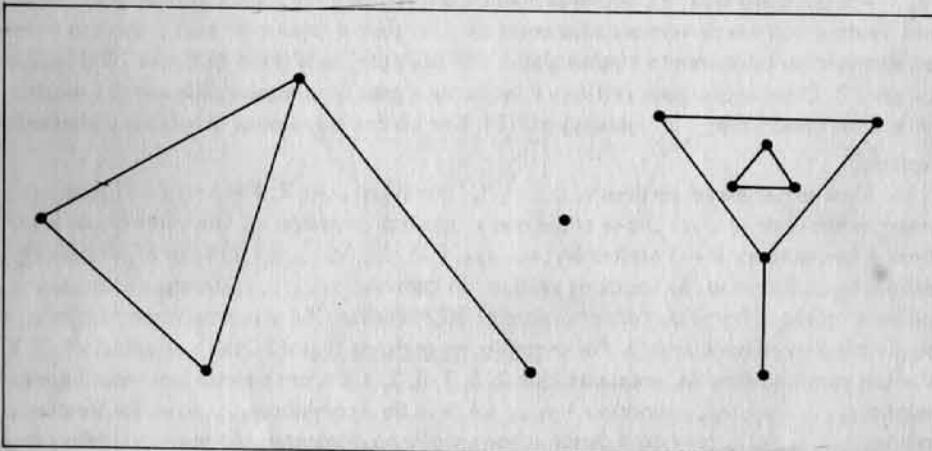


Figura 2.4. Um grafo desconexo

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 $e \in E$ 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 em 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 .

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 constitui o primeiro teorema conhecido em Teoria de Grafos, mencionado no capítulo 1 e reproduzido abaixo.

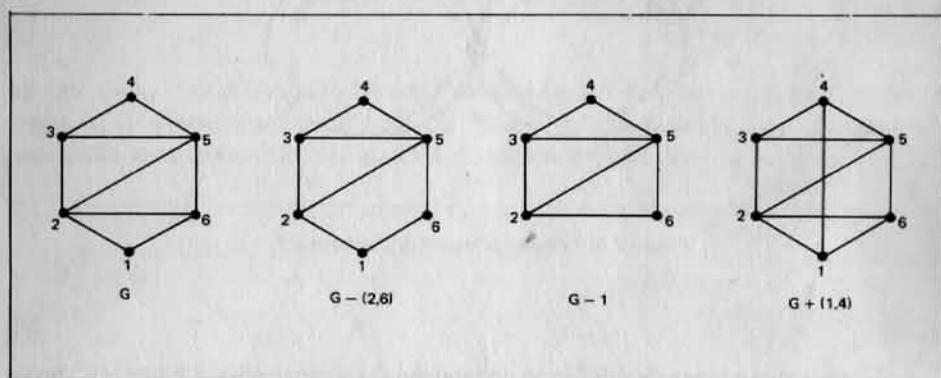


Figura 2.5. Operações de exclusão e inclusão de arestas ou vértices

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 cômputo 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 mo-

do. Como cada vértice de G possui grau ≥ 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.

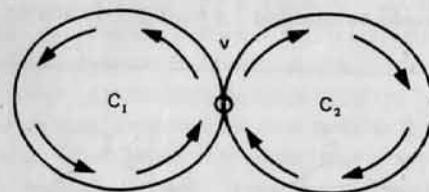


Figura 2.6. O passo principal do teorema 2.1

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 *bipartite* quando o seu conjunto de vértices V puder ser particionado 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 bipartite G por $(V_1 \cup V_2, E)$. Um grafo *bipartite completo* possui uma aresta para cada par de vértices v_1, v_2 , sendo $v_1 \in V_1$ e $v_2 \in V_2$. Sendo

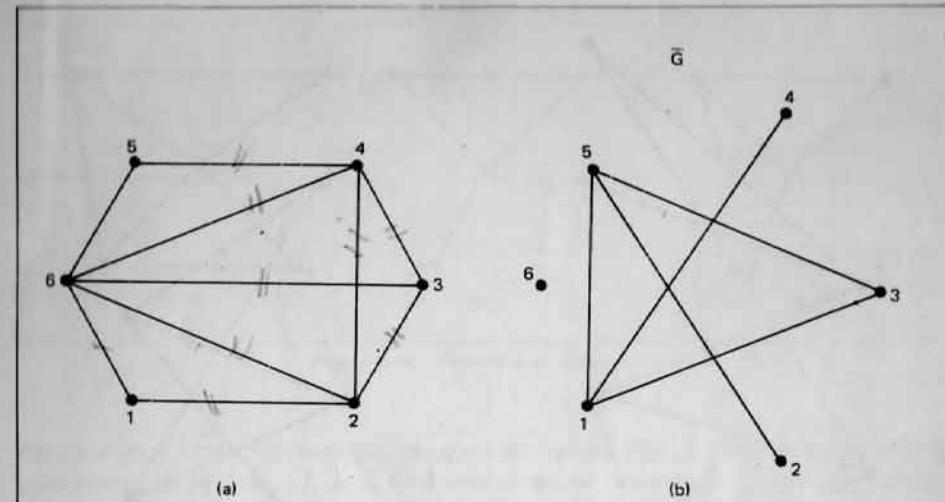


Figura 2.7. Um grafo e seu complemento

$n_1 = |V_1|$ e $n_2 = |V_2|$, um grafo bipartite completo é denominado 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 bipartites. A resposta é afirmativa.

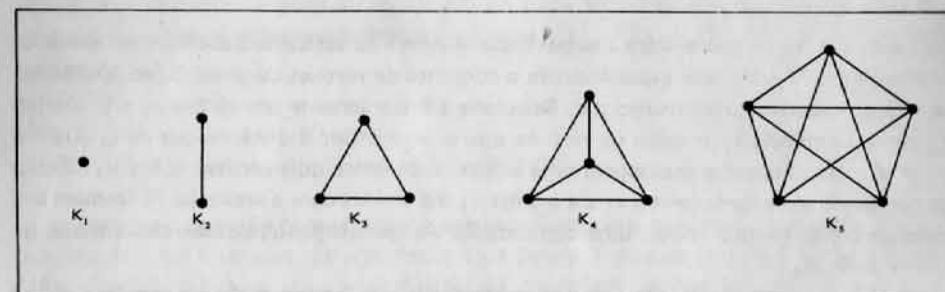


Figura 2.8. Grafos completos

Teorema 2.2

Um grafo $G(V, E)$ é bipartite 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 bipartite 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 $v_k \in V_2$,

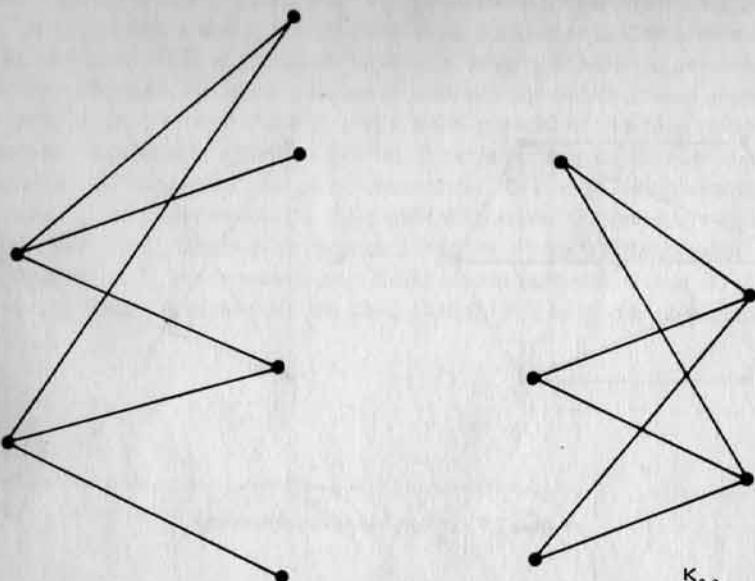


Figura 2.9. Exemplos de grafo bipartite

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. Selecione 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: 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.

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

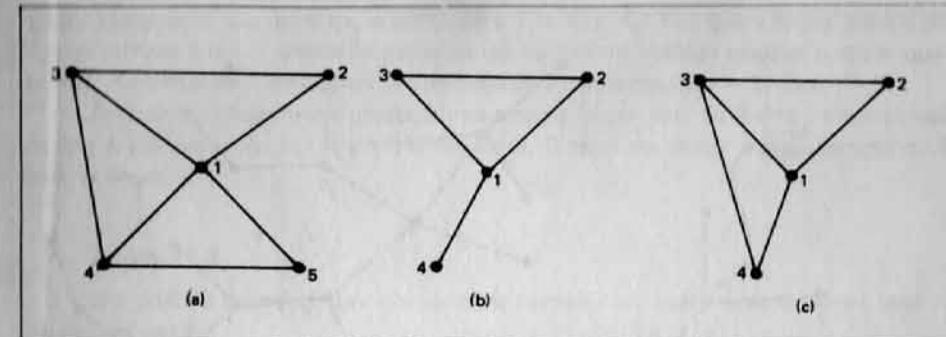


Figura 2.10. Exemplos de subgrafos

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, examinando o subgrafo induzido de cada um dos $\binom{n}{k}$ 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 ≤ 1 então v é uma *folha*. Caso contrário, se 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 isomórfas entre si) com 6 vértices.

Observe que toda árvore T com n vértices possui exatamente $n - 1$ arestas. O seguinte argumento indutivo justifica a afirmativa. Se T possuir 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 o pertence (caso contrário, a inclusão de uma nova aresta entre dois vértices de T produz um ciclo).

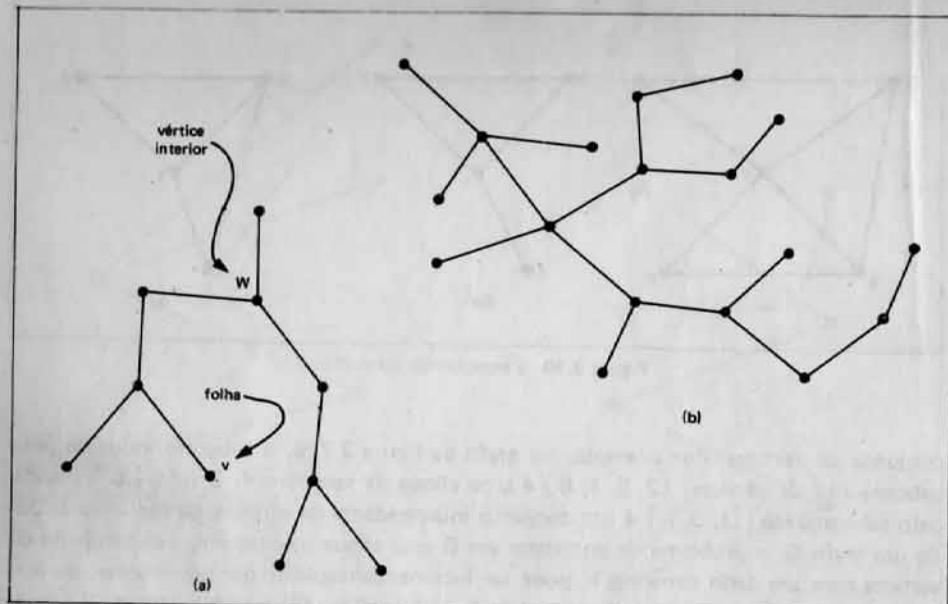


Figura 2.11. Uma floresta composta de duas árvores

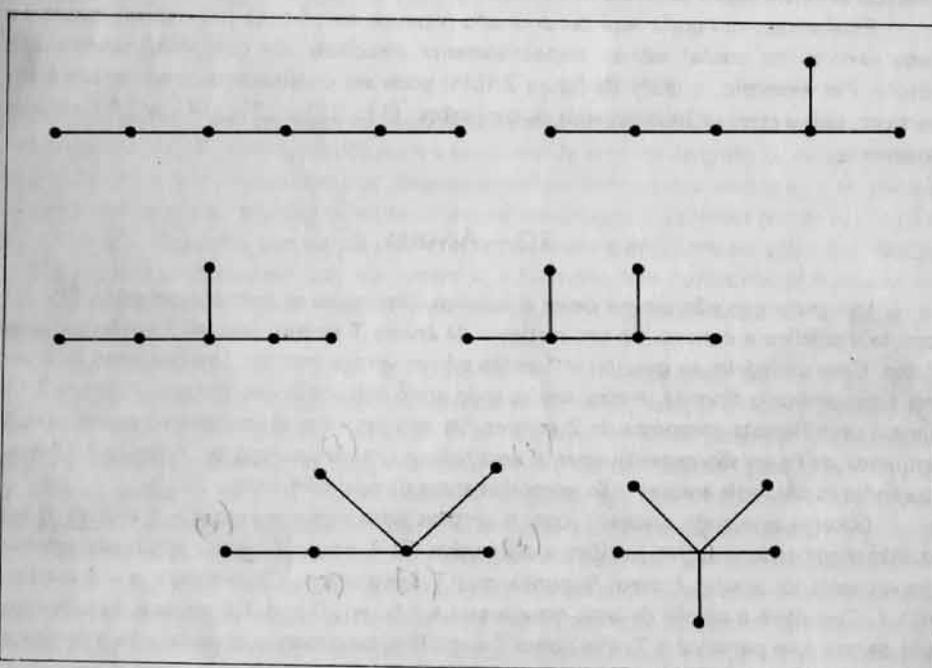


Figura 2.12. As árvores com 6 vértices

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.

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 , então 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 obviamente conexo e, além disso, não pode conter ciclos. Logo G é uma árvore. ▲

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

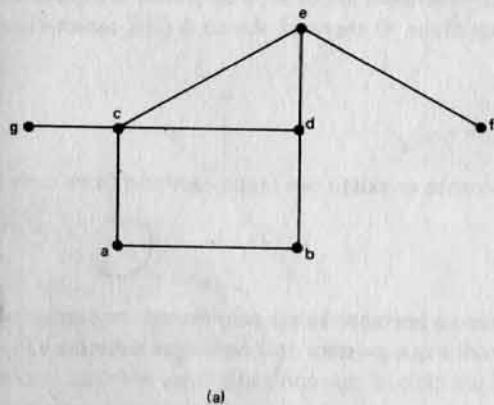
Teorema 2.4

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 acima não apresenta dificuldade e será omitida. Seja $G(V, E)$ um grafo. Denomina-se *excentricidade* de um vértice $v \in V$ ao valor da distância máxima 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.



(a)

vértice	excentricidade
a	3
b	3
c	2
d	2
e	2
f	3
g	3

(b)

Figura 2.13. Excentricidade de vértices

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. \blacktriangleleft

Tem-se então:

Teorema 2.4

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 2.4 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 acima, 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 idéia a cada componente conexo de G , obtendo então uma *floresta geradora* de G , ou seja, uma árvore geradora para cada componente conexo do grafo.

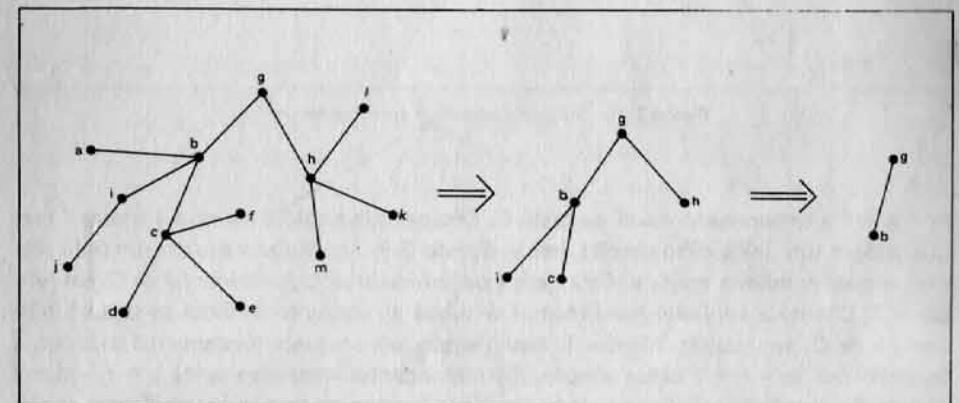


Figura 2.14. Determinação do centro de uma árvore

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 à T . Sendo $|V| = n$, $|E| = m$ tem-se: $|E_T| = n - 1$ e portanto o número total de elos distintos de G é igual a $m - n + 1$. O valor

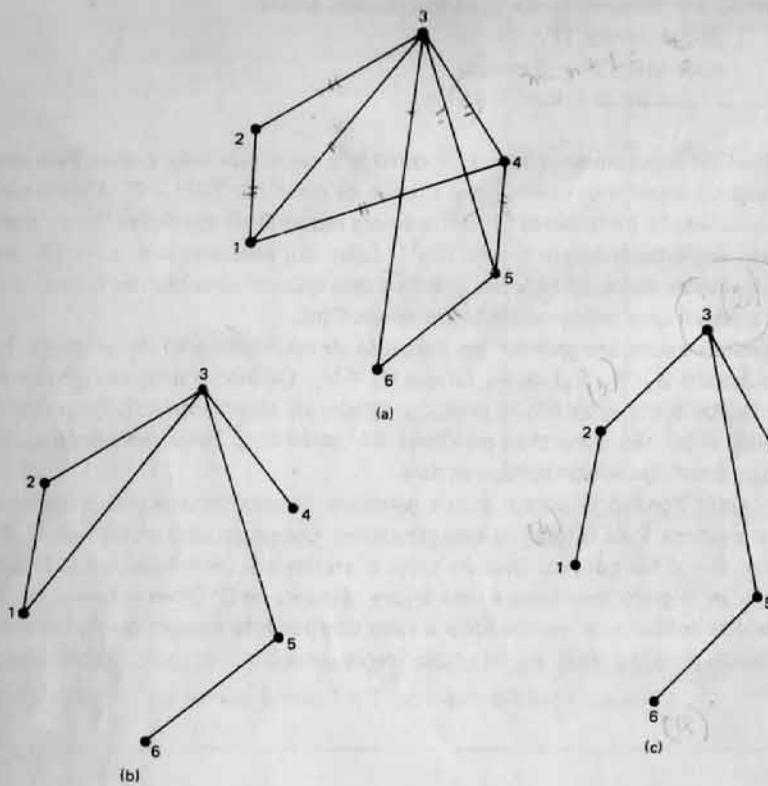


Figura 2.15. Subgrafo gerador e árvore geradora

$m - n + 1$ 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. Este vértice é então 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*.

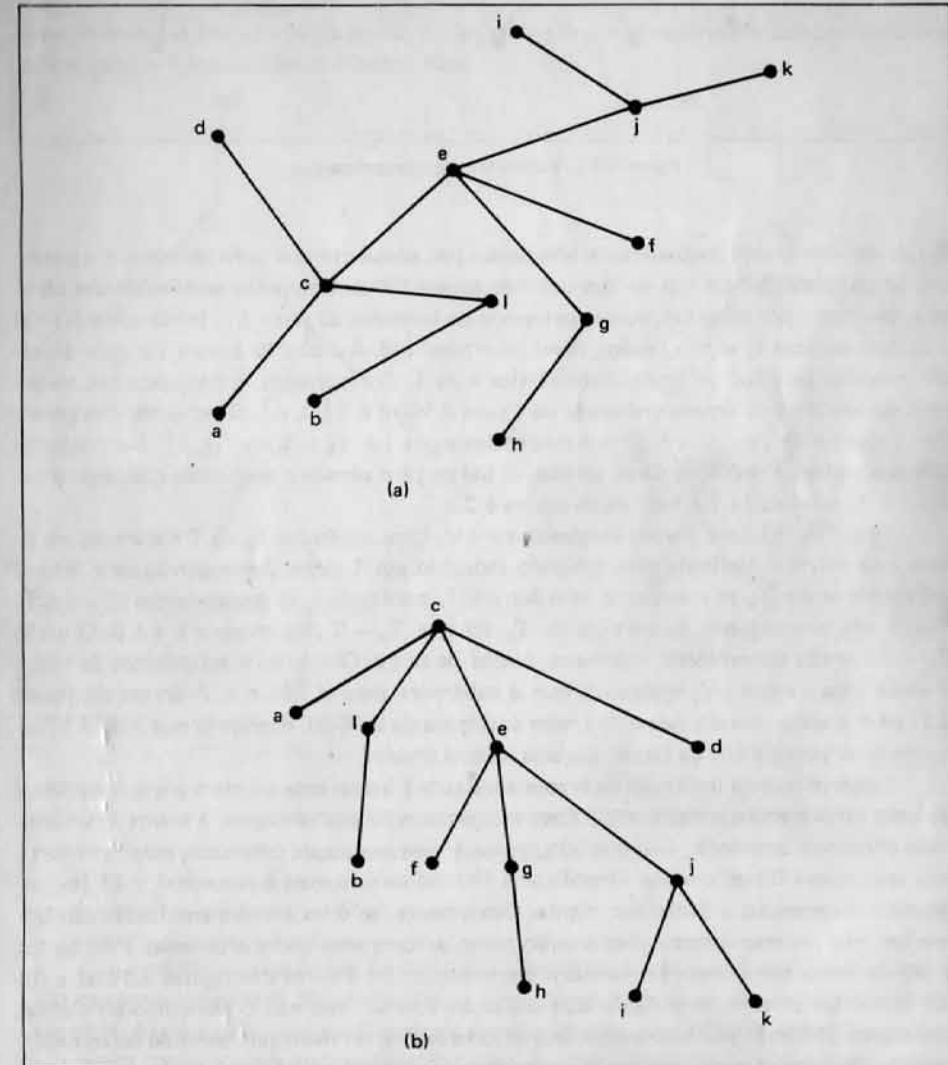


Figura 2.16. Árvores

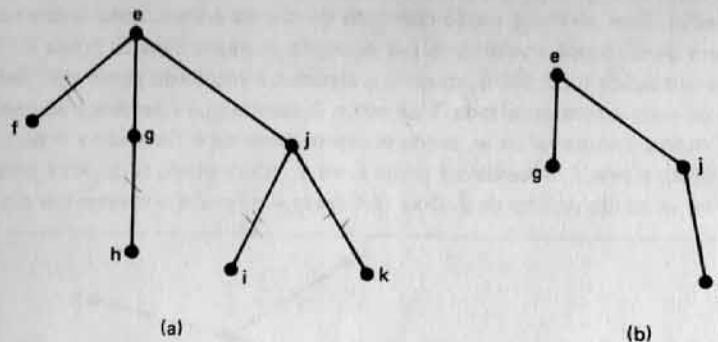


Figura 2.17. Subárvore e subárvore parcial

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 comprimento do caminho da raiz r a v . Então $\text{nível}(r) = 0$ 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 $\{e, c\}$. O dos 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.

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 u 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 uma 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 1º filho de v (o mais à esquerda), o 2º (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 .

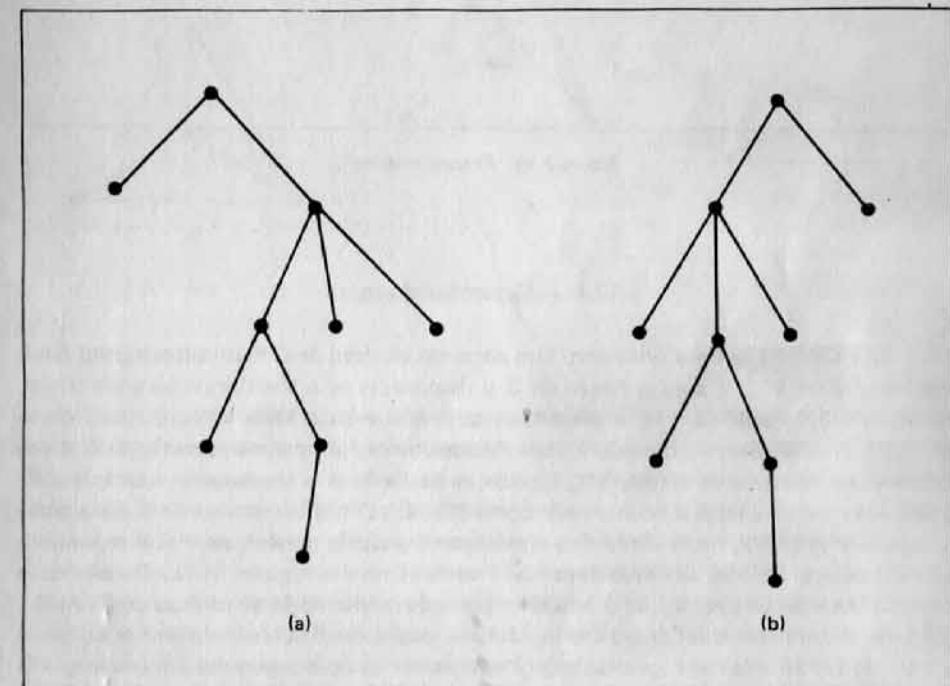


Figura 2.18. Árvores enraizadas isomórficas

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.

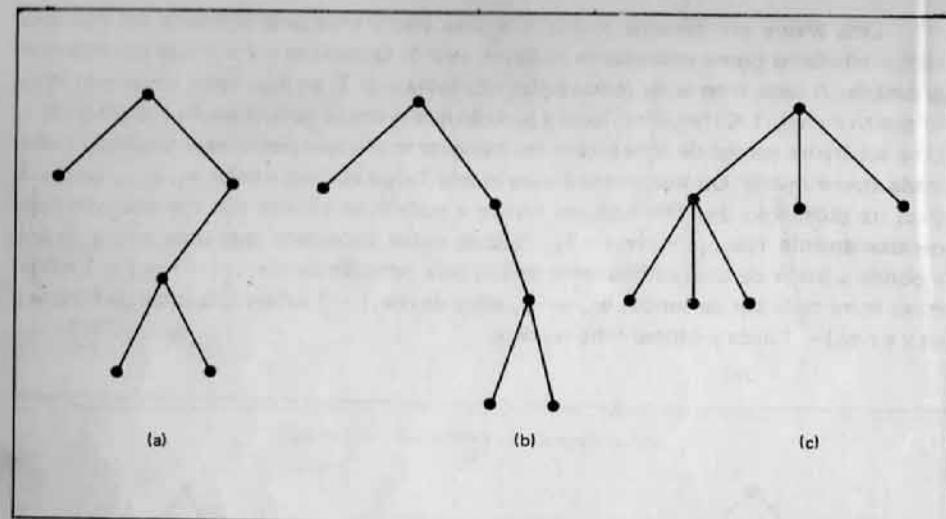


Figura 2.19. Árvores m-árias

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 nos componentes ilustrados 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 são respectivamente iguais a 1 e 2.

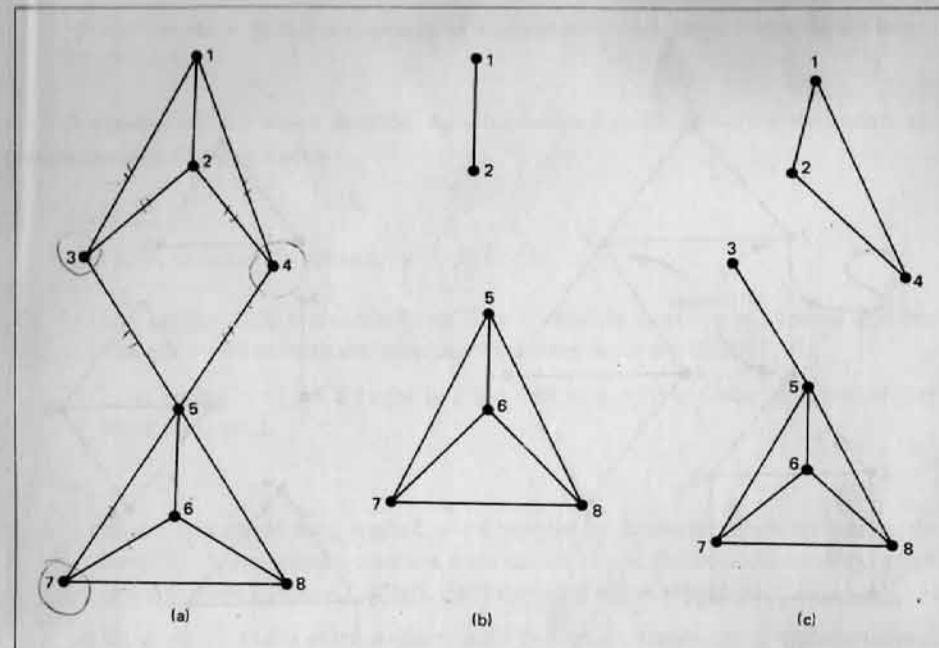


Figura 2.20. Cortes de um grafo

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), por exemplo, é 1-conexo em vértices, 4-conexo 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 em E' é incidente a algum $v \in V'$. Conseqüentemente, 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-se do grafo as $|g(w)|$ arestas adjacentes a w . Então $g(w) \geq c_E \geq c_V$. Observe que se G for o grafo completo K_n , então $g(w) = c_E = c_V = n - 1$.

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. Seja $G(V, E)$ um grafo. Um vértice v é denominado *articulação* quando sua remoção de G o desconecta. Ou seja, $G - v$ é desconexo. Uma aresta e 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 biconexos* do

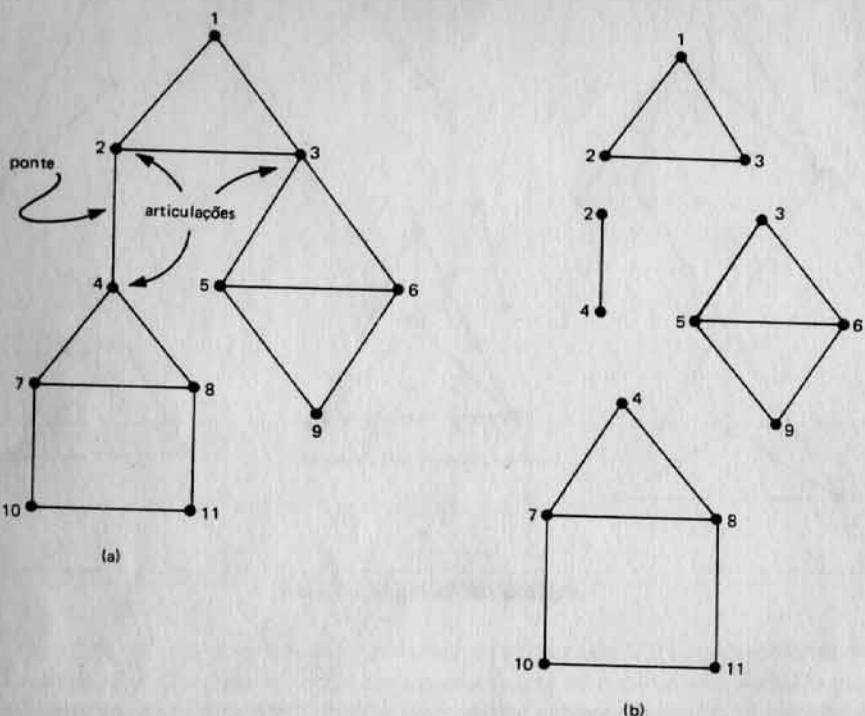


Figura 2.21. Um grafo e seus blocos

grafo G aos subgrafos máximos de G que sejam biconexos em vértices, ou isomorfos a K_2 . Cada componente biconexo é também chamado *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. Seus componentes biconexos estão indicados na figura 2.21(b). O lema seguinte traz alguma informação adicional sobre os blocos de um grafo.

Lema 2.2

Seja G 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 abaixo.

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 conexos distintos 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 conexos distintos 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.

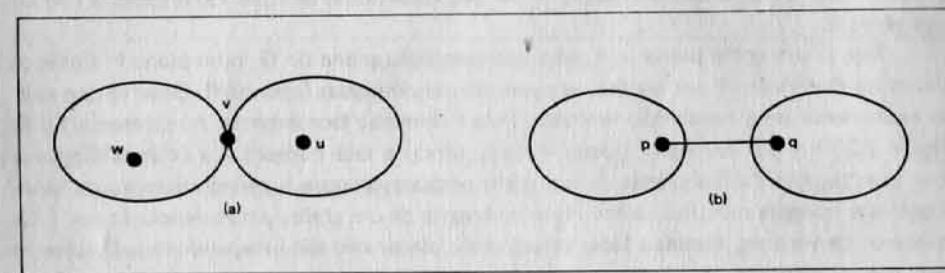


Figura 2.22. Prova do lema 2.3

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 acima pode ser generalizado, como se segue.

Teorema 2.5

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 acima 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 2.6

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 2.5 e 2.6 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 2.7

Seja G um grafo planar. Então $n + f = m + 2$.

Uma conveniente representação plana de G corresponde a um poliedro com n vértices, m arestas e f faces. A expressão acima é pois 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.

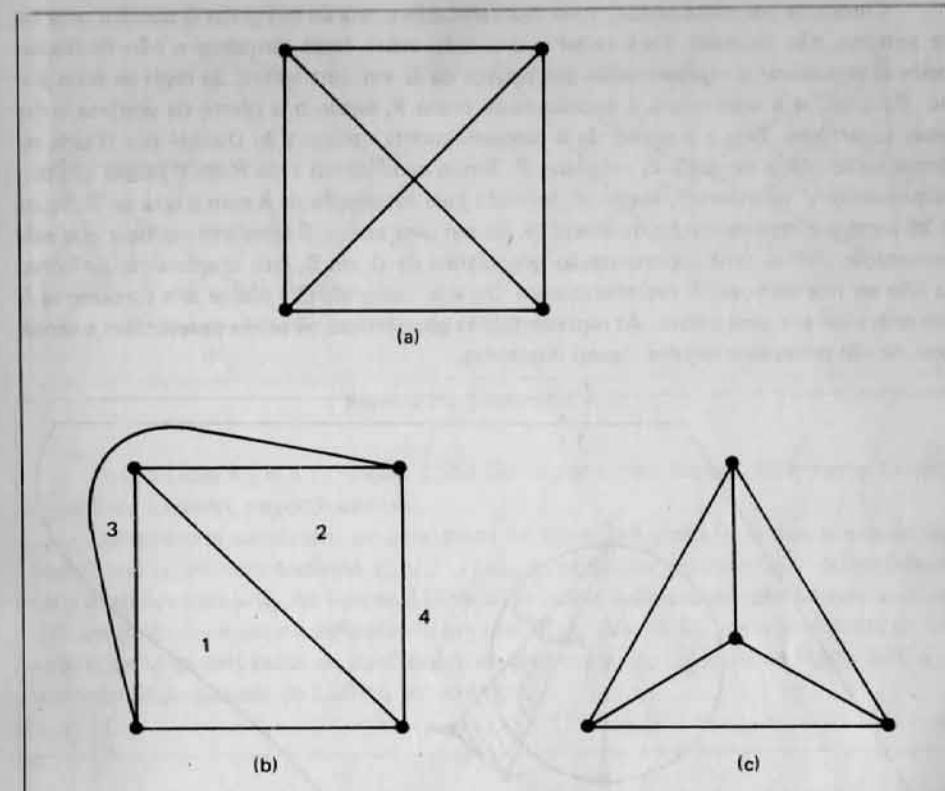


Figura 2.23. Um grafo planar

Lema 2.5

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

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:

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

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 são retas. 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.

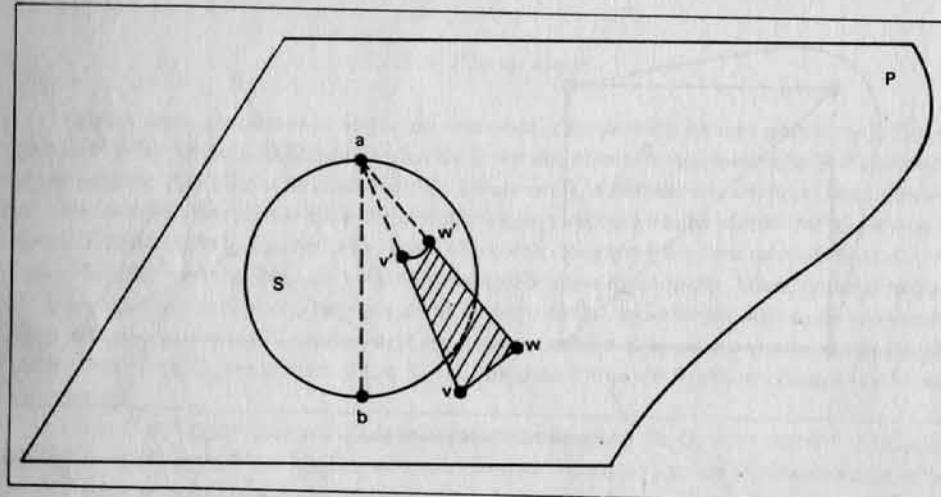


Figura 2.24. Imersão de um grafo planar na esfera

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

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 $m = 9$, $f = 5$.

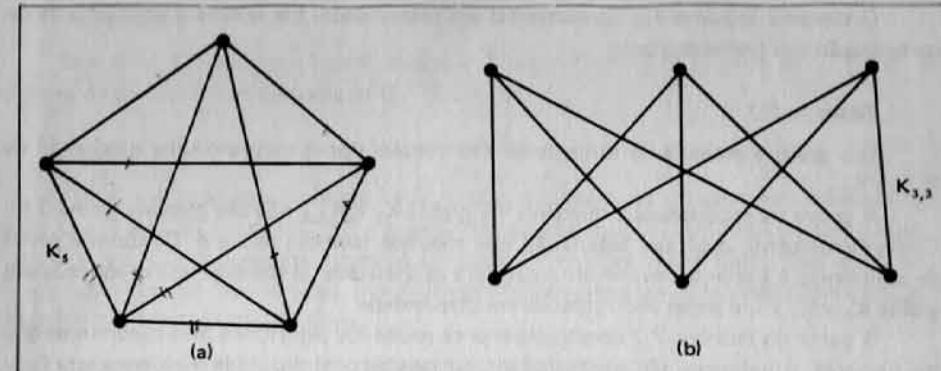


Figura 2.25. Os grafos K_5 e $K_{3,3}$

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 seqüência de subdivisões de arestas de G_1 . O grafo da figura 2.26(d) é uma subdivisão daquele da 2.26(c), por exemplo.

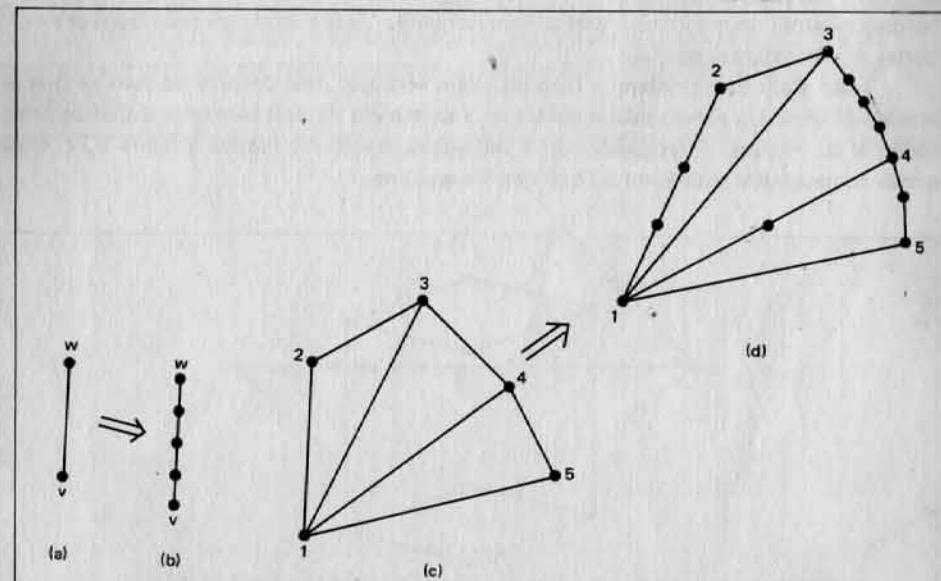


Figura 2.26. Subdivisão de arestas e grafos

O teorema seguinte é o fundamental em planaridade. Ele resolve o problema de caracterização dos grafos planares.

Teorema 2.7

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). Conseqüentemente qualquer subdivisão dos mesmos também não o é. Contudo a prova de suficiência é bastante envolvente e não será apresentada. O teorema acima concede aos grafos K_5 e $K_{3,3}$ um papel todo especial em planaridade.

A partir do teorema 2.7 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.

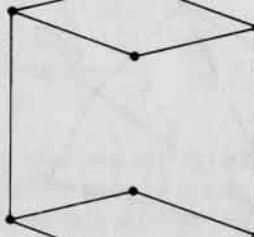


Figura 2.27. Um grafo biconexo não hamiltoniano

Teorema 2.8

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

Prova

Seja C um ciclo hamiltoniano de G . Então o número de componentes conexos do grafo $C - S \leq |S|$ (porque $C - S$ é a união de, no máximo, $|S|$ caminhos simples disjuntos). $C - S$ 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 conexos desse último é \leq ao do primeiro. Isto prova o teorema. Δ

Uma condição suficientemente clássica é a seguinte:

Teorema 2.9

Seja $G(V, E)$ um grafo com pelo menos 3 vértices e tal que 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 grau (v) , 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 G 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. Δ

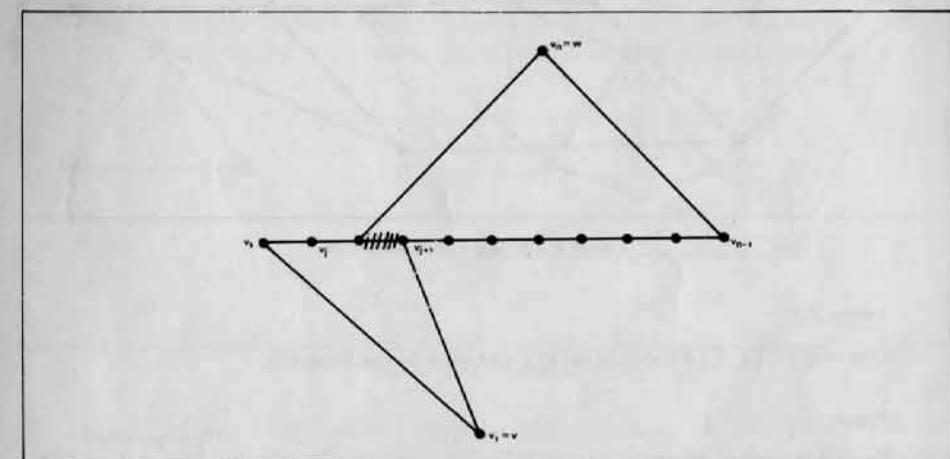


Figura 2.28. Prova do lema 2.9

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 f cuja cardinalidade do conjunto imagem é igual a k . Diz-se então que G é k -*colorível*. Denomina-se *número cromático* $X(G)$ 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 7) de que não existe algoritmo desta natureza. Contudo, o caso especial de bicoloração, pode ser resolvido de forma simples.

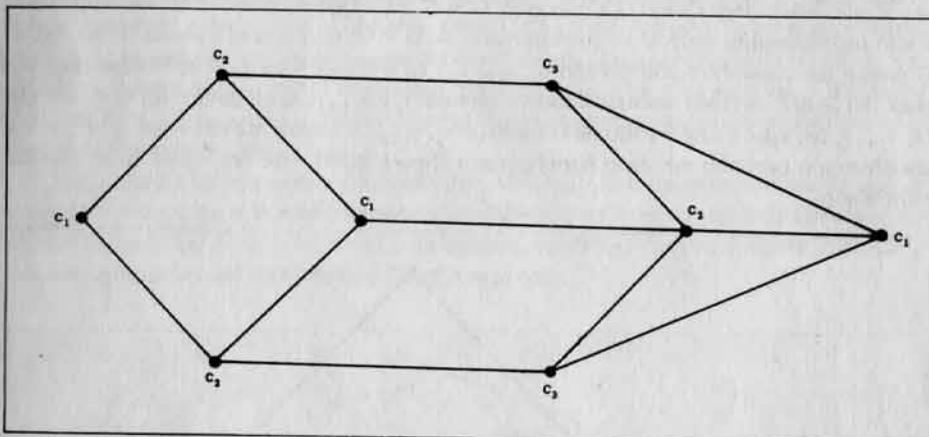


Figura 2.29. Coloração de grafos

Lema 2.7

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

Prova

Se G é bipartite 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 bico-

lorível, 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 acima 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. Então obviamente $\cup V_i = V$ 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 particionar V em um número mínimo de conjuntos independentes de vértices.

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

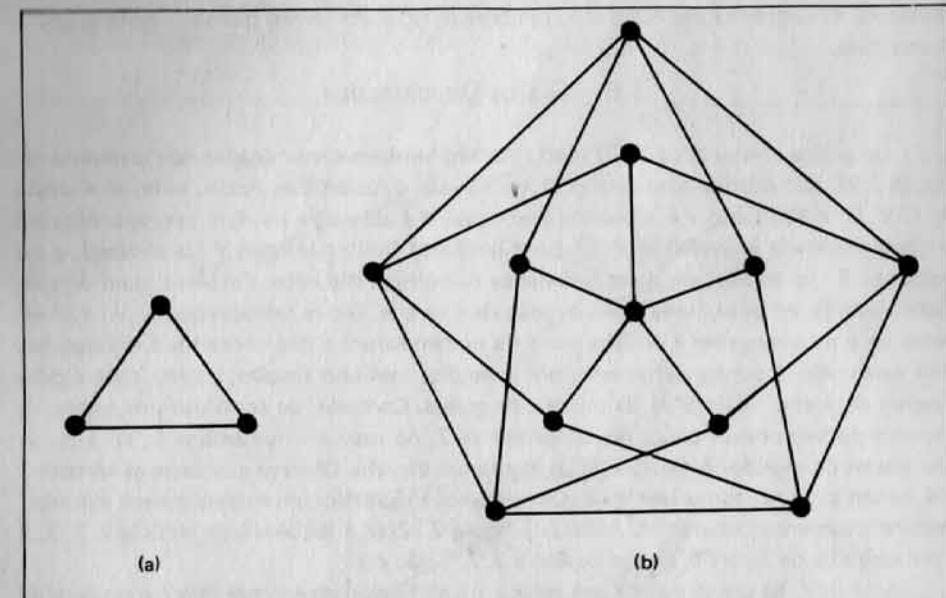


Figura 2.30. Grafos k -críticos

Teorema 2.10

Se $G(V, E)$ é k -crítico então $\deg(v) \geq k - 1$, 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 $(k - 1)$ -coloração de G , uma contradição.

O estudo de coloração foi iniciado com o *problema das quatro cores*, conforme mencionado na seção 1.1. Seja M uma região do plano. Um *mapa* é um particionamento de M em um número finito de sub-regiões, as quais são possivelmente 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 adjacentes 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.

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* (*dígrafo*) $D(V, E)$ é um conjunto finito não vazio V (os *vértices*), e um conjunto E (as *arestas*) de pares ordenados de vértices distintos. Portanto, num dígrafo 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, ciclo simples de forma análoga às definições de grafos. Contudo, ao contrário dos grafos, os dígrafos podem possuir ciclos de comprimento 2, no caso em que ambos (v, w) e (w, v) são arestas do dígrafo. A figura 2.32(a) ilustra um dígrafo. Observe que entre os vértices 3 e 4 há um ciclo de comprimento 2. Os caminhos e ciclos em um dígrafo devem obedecer ao direcionamento das arestas. Assim, na figura 2.32(a), a seqüência de vértices 2, 7, 3, 5 é um caminho de 2 para 5, enquanto que 2, 7, 5 não o é.

Seja $D(V, E)$ um dígrafo 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.

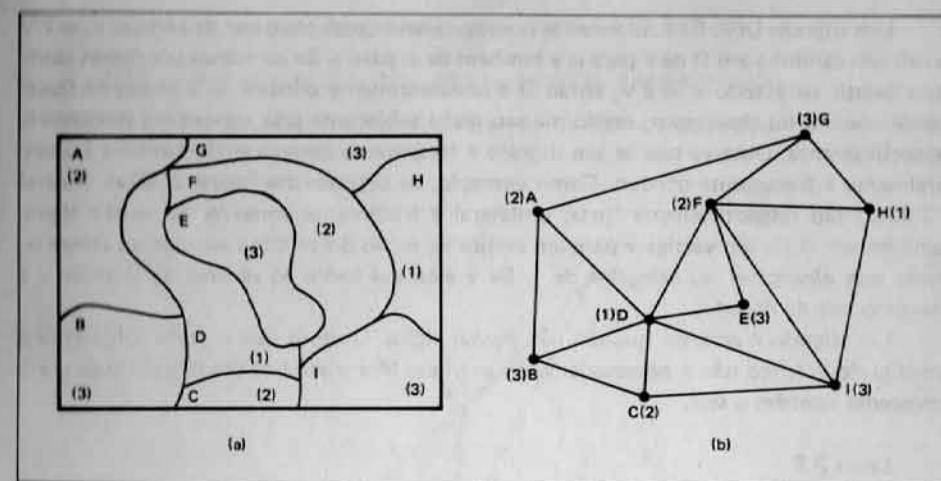


Figura 2.31. Coloração de mapas

Se forem retiradas as direções das arestas de um dígrafo 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 dígrafo 2.32(a).

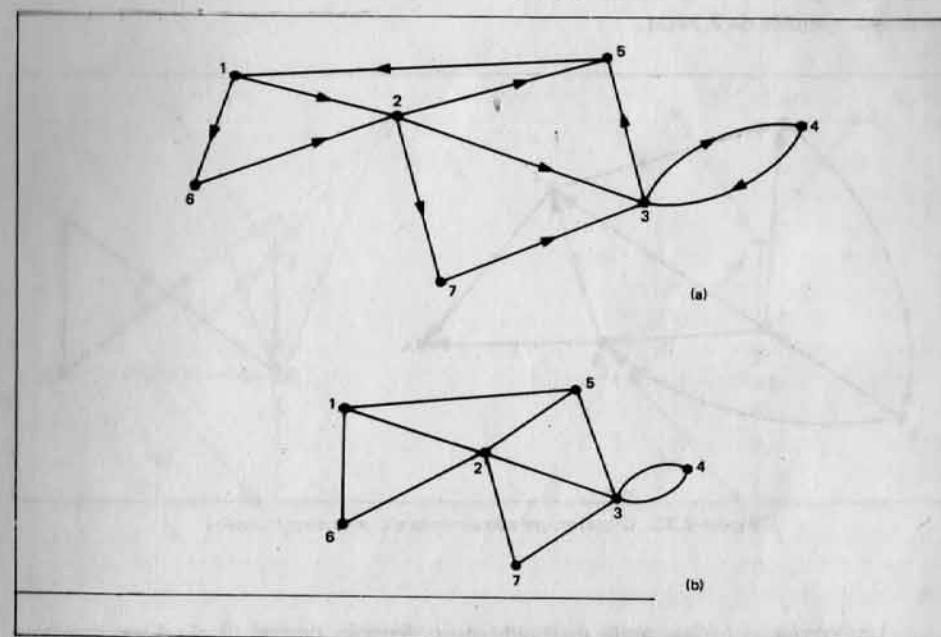


Figura 2.32. Um dígrafo e seu grafo subjacente

Um dígrafo $D(V, E)$ 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*. D é chamado *fracamente conexo* ou *desconexo*, conforme seu grafo subjacente seja conexo ou desconexo, respectivamente. Observe que se um dígrafo é fortemente conexo então também é unilateralmente e fracamente conexo. Como exemplo, os dígrafos das figuras 2.32(a), 2.33(a) e 2.33(b) são respectivamente forte, unilateral e fracamente conexos. Se existir algum caminho em D de um vértice v para um vértice w , 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 dígrafo.

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

Lema 2.8

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

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

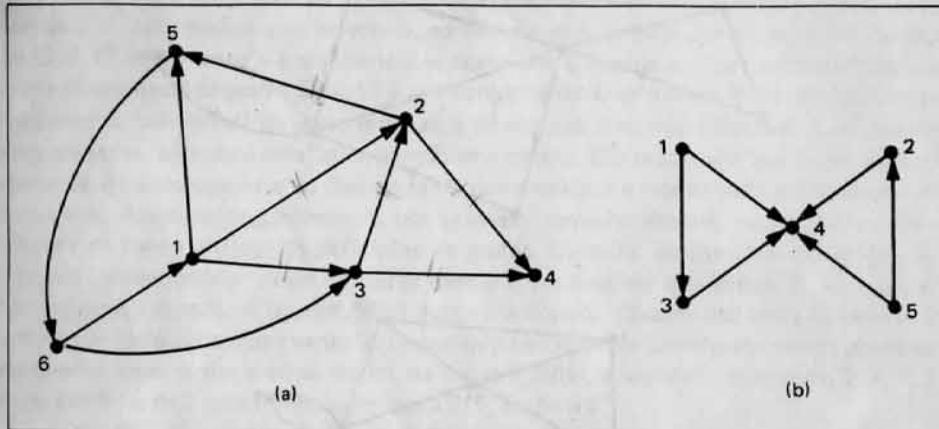


Figura 2.33. Dígrafos unilateralmente e fracamente conexos

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).

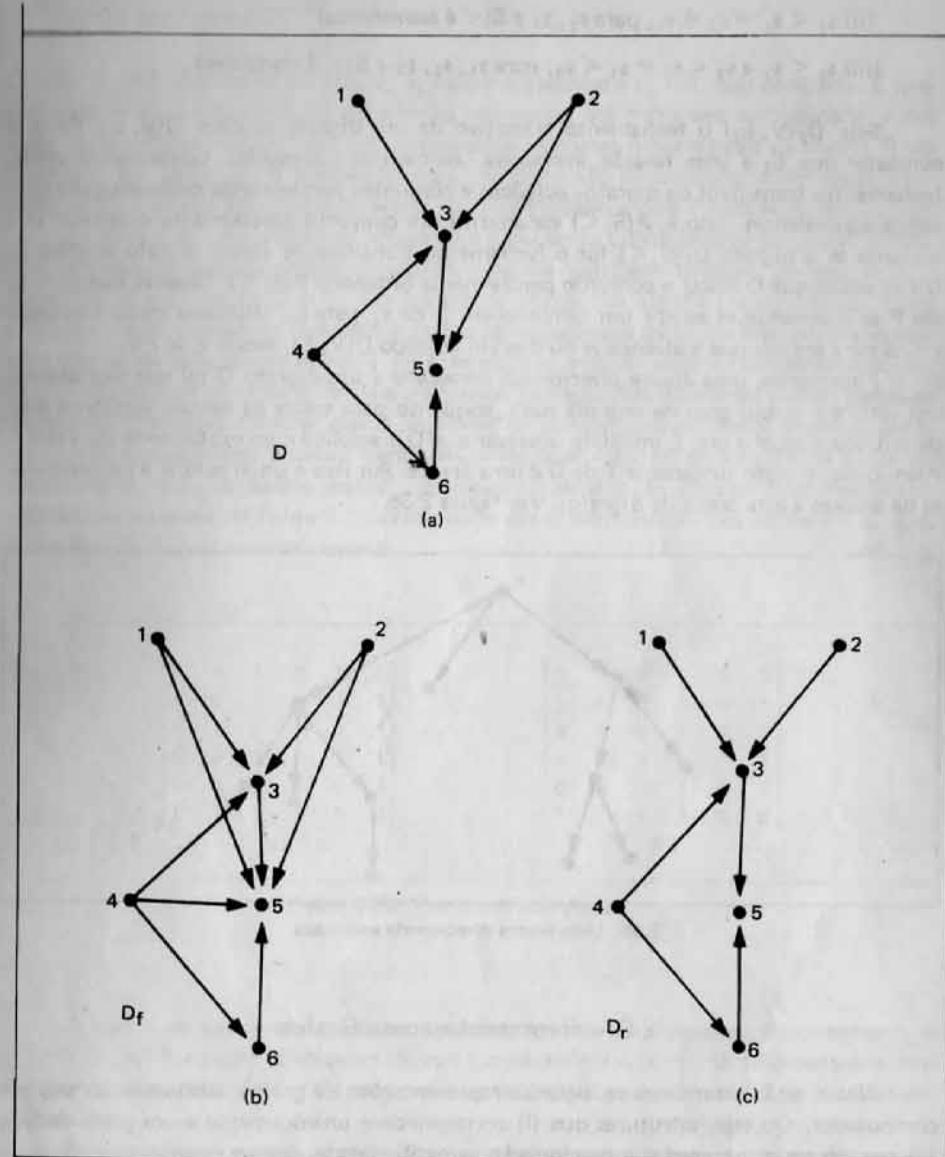


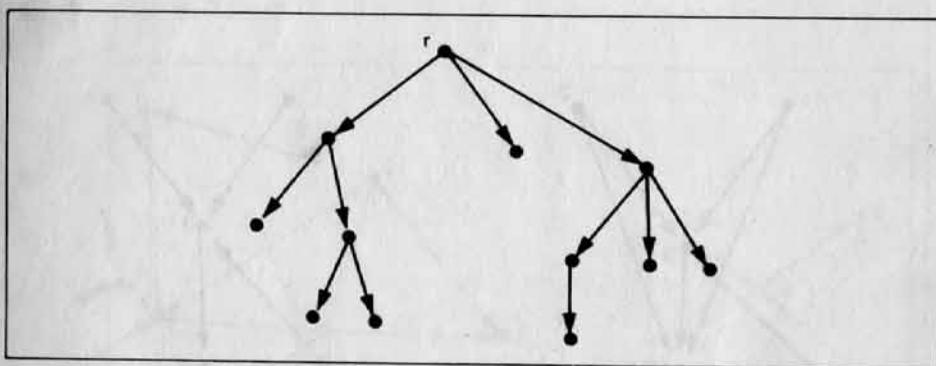
Figura 2.34. Um dígrafo, seu fechamento e redução transitivos

Um conjunto parcialmente ordenado (S, \leq) pode ser também caracterizado pelo par $(S, <)$ onde $<$ é uma relação binária em S definida por $s_1 < s_2 \iff s_1 \leq s_2 \text{ e } s_1 \neq s_2$. A relação $<$ satisfaz:

- (i) $s_1 < s_1$, para $s_1 \in S$ ($<$ é irreflexiva)
- (ii) $s_1 < s_2 \Rightarrow s_2 < s_1$, para $s_1, s_2 \in S$ ($<$ é assimétrica)
- (iii) $s_1 < s_2 \text{ e } s_2 < s_3 \Rightarrow s_1 < s_3$, para $s_1, s_2, s_3 \in S$ ($<$ é transitiva).

Seja $D_f(V, E_f)$ o fechamento transitivo de um dígrafo acíclico $D(V, E)$. Pode-se constatar que E_f é uma relação irreflexiva, assimétrica e transitiva. Conseqüentemente, fechamentos transitivos de dígrafos acíclicos e conjuntos parcialmente ordenados são conceitos equivalentes. Isto é, $P(S, <)$ caracteriza um conjunto parcialmente ordenado se e somente se o dígrafo $D_f(S, <)$ for o fechamento transitivo de algum dígrafo 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 dígrafo acíclico $D(V, E)$, sendo $v, w \in V$.

Finalmente, uma árvore direcionada enraizada é um dígrafo 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 dígrafos. Ver figura 2.35.



2.35. Uma árvore direcionada enraizada

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 vá-

rios 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:

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

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

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 ambigüidade. 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, 6, 6 e 5, 4, 1, 3, 2, 6, respectivamente.

$\begin{array}{ccccccc c} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{array}$	$\begin{array}{ccccccc c} 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{array}$
(a)	(b)

Figura 2.36. Matrizes de adjacências

A matriz de adjacências pode ser também definida para dígrafos. Basta tomar $r_{ij} = 1 \iff (v_i, v_j)$ 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 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* $n \times m B = (b_{ij})$ assim definida:

$b_{ij} = 1 \iff$ 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). A complexidade de espaço da matriz de incidências é $O(nm)$, maior ainda do que a da matriz de adjacências.

adjacências.

1	1	1	1	0	0	0	0	0	0	0	0
1	0	0	0	1	1	0	0	0	0	0	0
0	1	0	0	1	0	1	1	0	0	0	1
0	0	0	0	0	0	0	0	1	0	1	1
0	0	0	1	0	0	1	0	1	1	0	0
0	0	1	0	0	1	0	1	1	0	0	0

Figura 2.37. Matriz de incidências

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 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 de 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, $p(v) = \phi$. Pode-se concluir então que o espaço utilizado por uma estrutura de 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

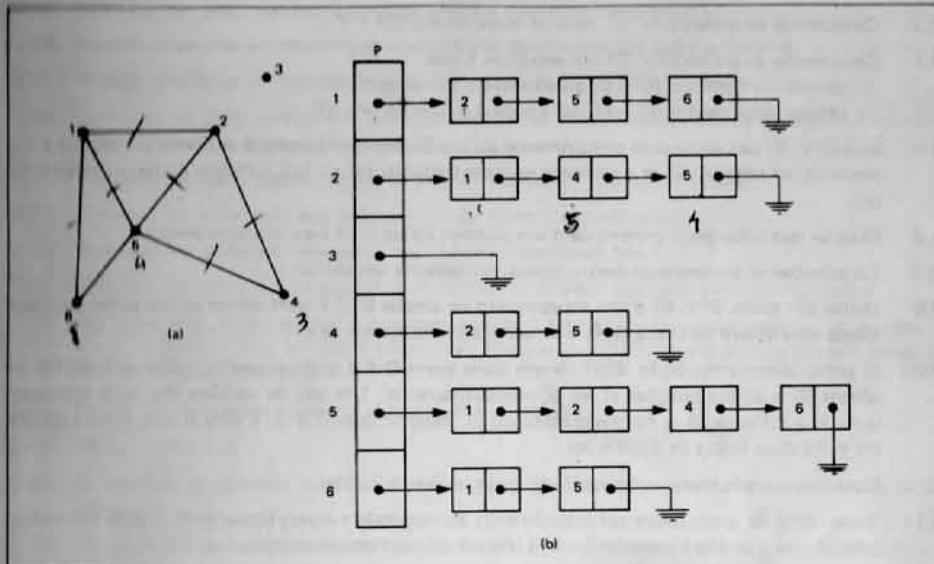


Figura 2.38. Estrutura de adjacência

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 dígrafos de forma análoga. Isto é, para um dígrafo $D(V, E)$ 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 se verificar que o espaço requerido pela estrutura A para representar o dígrafo D é também linear com o tamanho de D .

2.10 – EXERCÍCIOS

- 2.1 Construir uma representação geométrica do grafo:

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

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

- 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 $s \in V -$ centro (G).
 (ii) centro (G) = centro ($G - S$), para todo $S \subseteq V -$ 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 bipartite, cujos vértices são os blocos B_j 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_j . 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.

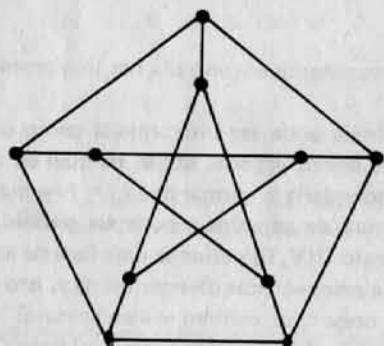


Figura 2.39. O grafo de Petersen

- 2.12 Mostrar que o *grafo de Petersen*, ilustrado na figura 2.39, 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 bipartite 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 2.8 não é suficiente.
- 2.19 Mostrar através de um exemplo que a condição do teorema 2.9 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 $v, w \in V$, o grafo $G + (v, w)$ não é k -colorível. Mostrar que G é maximal 2-colorível se e somente se G for bipartite completo.
- 2.26 Provar que todo grafo G admite subgrafo $X(G)$ -crítico.
- 2.27 Provar o lema 2.8.
- 2.28 Caracterizar os digrafos acíclicos D para os quais o fechamento 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 $S = (s_{ij})_{|V| \times |E|}$, como:
 $s_{ij} = 1 \iff$ as arestas e_i, e_j são incidentes
 $s_{ij} = 0$, caso contrário.
 Então S é uma representação de G . Provar ou dar exemplo.

2.11 – 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). Os seguintes são livros sobre temas mais específicos. Tutte (1966), conectividade. Ore (1967) e Saaty e Kainen (1977), problema das quatro cores. Bollobás (1978), grafos extremos. Busacker e Saaty (1965) e Harary, Norman e Cartwright (1965), grafos direcionados. Golumbic (1980), 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. A literatura brasileira em grafos inclui os textos de Andrade (1980), Barbosa (1974), Boaventura (1979), Furtado (1973), Lucchesi (1979) e Savulescu (1980). Rabuske (1981) é um trabalho sobre hipergrafos em português. Como mencionado, os trabalhos pioneiros em árvores fo-

ram 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 (1968). O teorema 2.5 é de Dirac (1960), enquanto que o 2.6 é de Whitney (1932). Esta caracterização constitui uma variação do Teorema de Menger (1927). Diversas outras variações desse teorema foram desenvolvidas. O teorema 2.7, 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 2.9 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 2.8 a 2.10 foram escritas a partir de Bondy e Murty (1976). 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 dígrafos torneios, descrito em Redei (1934).

CAPÍTULO 3

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 freqüência 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 dígrafo, 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 dígrafo 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 ϕ . A estrutura de dados necessária consiste pois de um vetor p com 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 dígrafo dado seja $V = \{1, 2, \dots, n\}$, sendo conhecido de antemão o seu número de arestas m .

*Motivos e algoritmos para ordenar
de adjacências de grafos comuns.*

algoritmo 3.1: Construção de uma estrutura de adjacências de um dígrafo $D(V, E)$

```

dados: n, m, E
para i = 1, ..., n efetuar p(i) := φ
para i = 1, ..., m efetuar
    seja (v, w) a i-ésima aresta de E
    ui := w
    ti := p(v)
    p(v) := i
  
```

Ao final do processo acima as listas de adjacências do dígrafo D estarão construídas. A complexidade desse algoritmo é obviamente $O(n + m)$. Deste ponto em diante do texto, convenciona-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 – Adjacência

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 algoritmos 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 idéia consiste em se examinar listas de adjacência $A(v)$, 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 Arestras

Esta técnica consiste na ordenação de vértices ou arestras 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. Arestras 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, freqüentemente 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 mí-

nimo 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 "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.

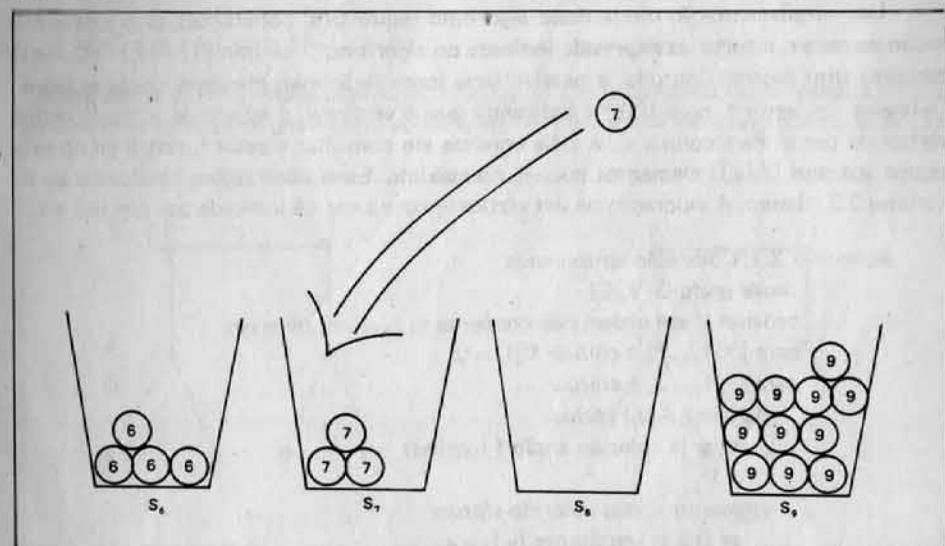


Figura 3.1. Ordenação por caixas

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 acima, 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

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

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 ($r := \min \{ i \mid A(v_j) \cap C_i = \emptyset \}$) 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 idéia 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, abaixo. A coloração de um vértice v_j com a cor r é indicada por $\text{cor}(v_j) = r$.

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

$r := 1$

enquanto v_j não colorido efetuar

se $f(r) \neq j$ então $\text{cor}(v_j) := r$

caso contrário $r := r + 1$

O número de operações efetuadas para colorir cada vértice v_j , no bloco definido por "para $j = 1, \dots, n$ efetuar" é $O(|A(v_j)|)$, pois necessariamente $f(i) \neq j$ 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. Porque 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.

3.6 – Ordenação Topológica

Conforme mencionado na seção 2.8, todo dígrafo acíclico $D(V, E)$ induz um conjunto parcialmente ordenado (V, \prec) , definido por:

$$v \prec w \iff v \text{ alcança } w \text{ em } D, \text{ para todo } v, w \in V$$

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

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

Tal seqüência é denominada *ordenação topológica*. Ela se constitui numa maneira natural de dispor os vértices de um dígrafo acíclico em uma linha reta, de modo que todas as suas

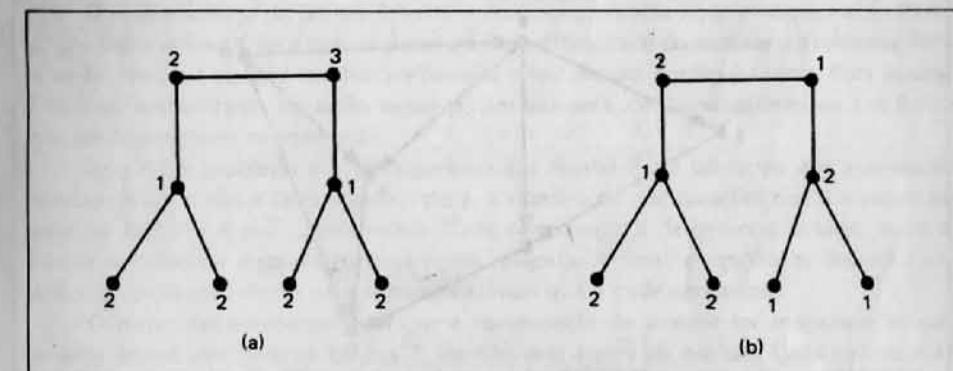


Figura 3.2. Colorações aproximada e ótima

arestas estejam direcionadas da esquerda para a direita. Vários algoritmos envolvendo dígrafos acíclicos possuem como passo intermediário a computação de uma ordenação topológica.

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

algoritmo 3.4: Ordenação topológica

dados dígrafo 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 dígrafo da figura 3.3, produz a ordenação topológica de f a b a c , supondo que na 3^a iteração fosse realizada a escolha $w = f$.

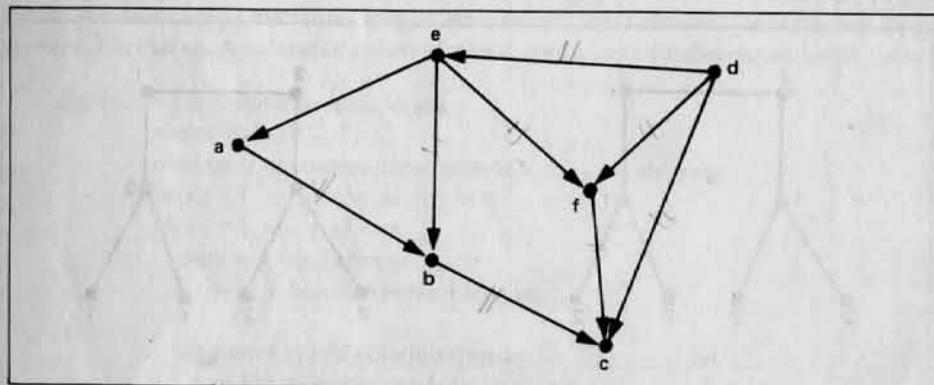


Figura 3.3. Exemplo para o algoritmo 3.3

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 dígrafo, 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 dígrafo.

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 decompostos 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:

```

factorial (n) := se n = 0 então 1
                  caso contrário n . factorial (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 α um algoritmo que resolve P , de tal modo que a operação dominante em α seja a *comparação*. Isto é, o número de comparações que α 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 α pode apresentar.

O efeito das comparações em uma computação de α 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 α , naturalmente. Um resultado verdadeiro de C, por exemplo, conduz α 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 α .

As folhas de T correspondem ao conjunto dos estados finais a que a computação de α pode conduzir. Ou seja, a cada possível saída diferente de α 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 α , para uma certa entrada E. Este número exprime a complexidade local assintótica de α , relativa a E. Conseqüentemente, a altura de T é igual ao número de comparações que α efetua no pior caso.

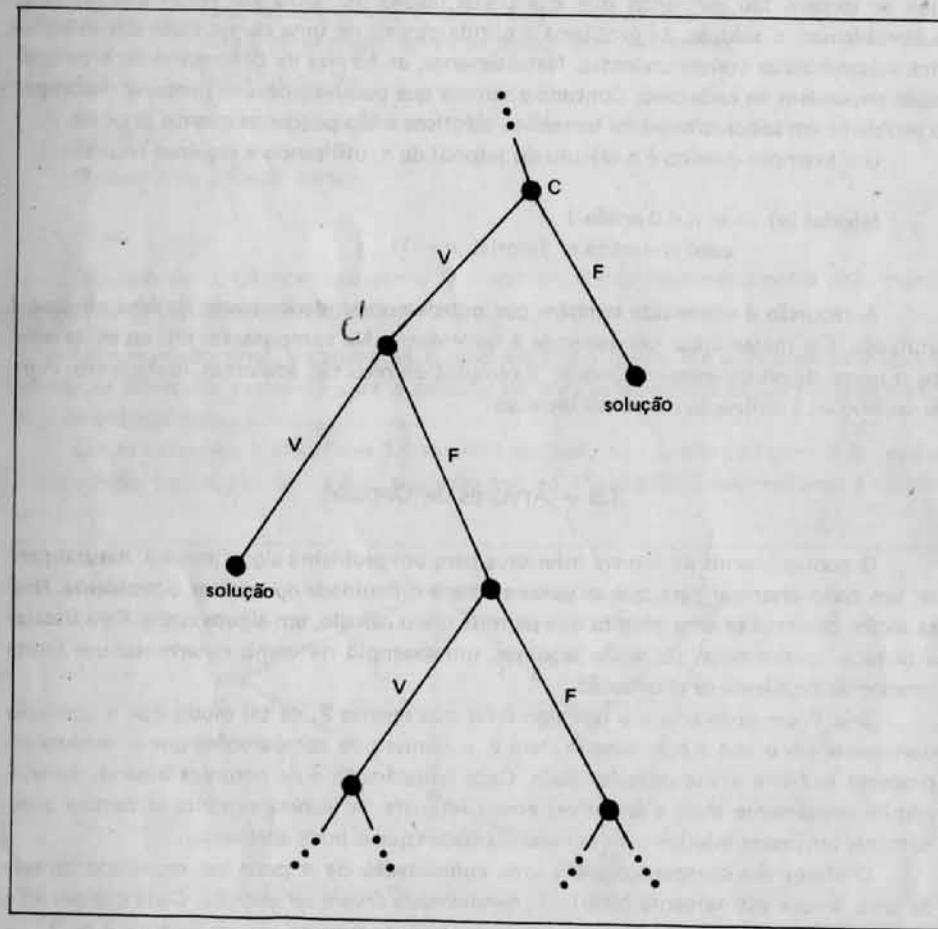


Figura 3.4. Árvore de Decisão

Ou seja, é igual à complexidade de α . 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 α 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 abaixo 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 α um algoritmo de ordenação nas condições acima e cuja entrada é S. Seja T a árvore de decisão de altura h, correspondente a α . O número de folhas de T é $\geq n!$, pois a saída de α pode corresponder a qualquer permutação de sua entrada. Assim sendo, T possui no máximo 2^h folhas, pois T é uma árvore binária. Logo,

$$2^h \geq n! \Rightarrow h \geq \log(n!). \text{ Como } n > 0,$$

$$n! = n(n-1)\dots 1 \geq n(n-1)\dots \lceil n/2 \rceil > (n/2)^{n/2}.$$

Conseqüentemente, $h > (n/2)(\log n - 2)$. 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. O algoritmo 1.4 da seção 1.3 satisfaz esta condição e possui complexidade $O(n \log n)$. Portanto, o limite inferior $\Omega(n \log n)$ é máximo e o algoritmo 1.4 é ótimo.

3.10 – EXERCÍCIOS

- 3.1 Modificar o algoritmo 3.1, de modo a construir uma estrutura de adjacências para um grafo não direcionado.
- 3.2 Descrever um algoritmo de ordenação por caixas, para realizar uma ordenação alfabética. Se existirem n itens a serem ordenados, cada qual com m caracteres alfabéticos, qual será a complexidade do processo?
- 3.3 Apresentar um exemplo de um grafo G, com número cromático X e tal que o algoritmo 3.2 de coloração aproximada atribua aos vértices de G um total de $2X$ ou mais cores.
- 3.4 Os algoritmos 3.2 e 3.3 produzem colorações rigorosamente iguais. Provar ou dar contra-exemplo.
- 3.5 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.

- 3.6 Um dígrafo admite ordenação topológica se e somente se for acíclico. Provar ou dar contra-exemplo.
- 3.7 Seja R uma matriz de adjacências de um dígrafo acíclico D , construída segundo uma permutação de seus vértices que corresponde a uma ordenação topológica. Mostrar que R é uma matriz triangular.
- 3.8 Seja $D(V, E)$ um dígrafo 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?
- 3.9 Seja T o número de ordenações topológicas distintas de um dígrafo $D(V, E)$. Quais são os valores mínimo e máximo de T ?
- 3.10 Seja A uma árvore direcionada enraizada. Determine o número de ordenações topológicas distintas dos vértices de A .
- 3.11 Seja T a árvore de decisão correspondente a um algoritmo α , baseado em ordenações, para resolver certo problema P . Então o nível mínimo de uma folha de T é igual ao número de comparações que α efetua no melhor caso. Provar ou dar contra-exemplo.

3.11 – 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). A produção de exemplos desfavoráveis para algoritmos aproximativos de coloração é tratado em Mitchem (1976). A dificuldade da obtenção de uma boa aproximação para este problema é apresentada em Garey e Johnson (1976). Um estudo de algoritmos de coloração é Santos (1979), em língua portuguesa. Um algoritmo de complexidade linear para o problema de ordenação topológica foi descrito inicialmente em Knuth (1968). Veja também Kahn (1962), Kase (1963) e Lasser (1961). A técnica da recursão é largamente empregada na formulação de algoritmos. Knuth (1968) 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).

CAPÍTULO 4

BUSCAS EM GRAFOS

4.1 – Introdução

Dentre as técnicas existentes para a solução de problemas algorítmicos em grafos, a busca ocupa 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:

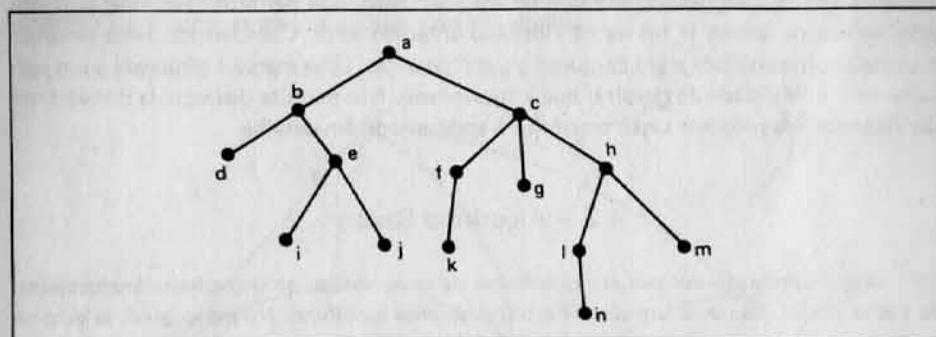


Figura 4.1. Uma árvore enraizada

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,
após pela 2º mais à esquerda,
após pela 3º 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 seqüência de visita:

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 seqüência 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 seqüência de visitas? 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

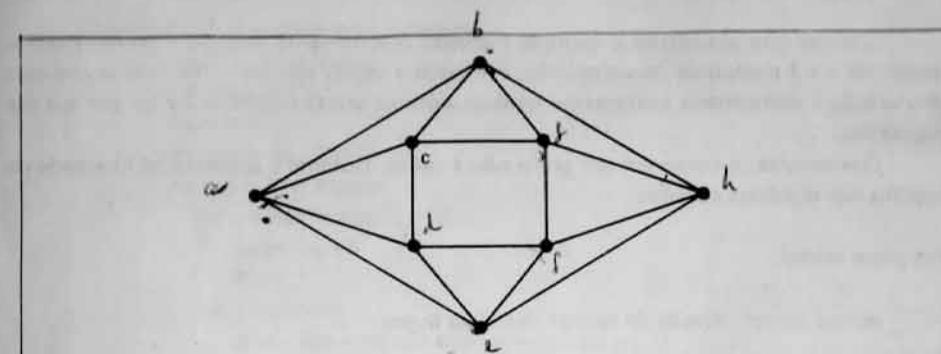


Figura 4.2. Como caminhar sistematicamente?

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 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).

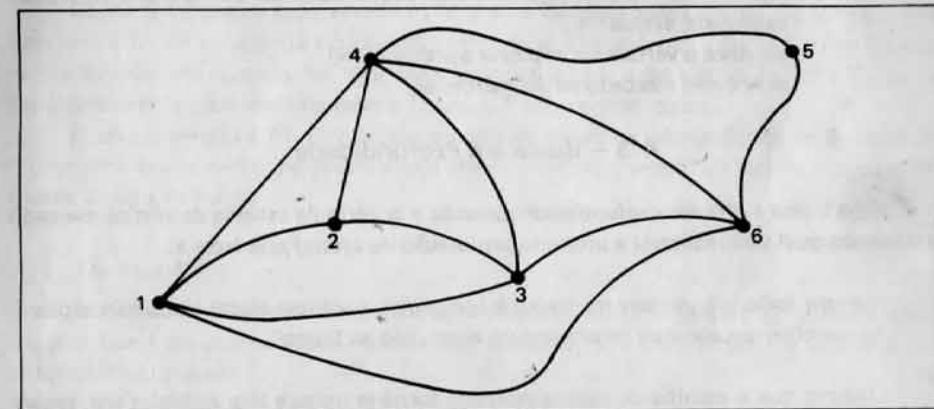


Figura 4.3. Exemplo para uma busca

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 .

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 ambigüidades.

algoritmo 4.1: Busca geral

dados grafo $G(V, E)$

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

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:

"dentre 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 ambigüidade, segundo o critério acima. O seguinte algoritmo recursivo implementa esse processo.

algoritmo 4.2: Busca em profundidade

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) (I)

$P(w)$

caso contrário

se $w \in Q$ e v, w não são consecutivos em Q

então visitar (v, w) (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 4.2 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 w 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.

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 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.

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

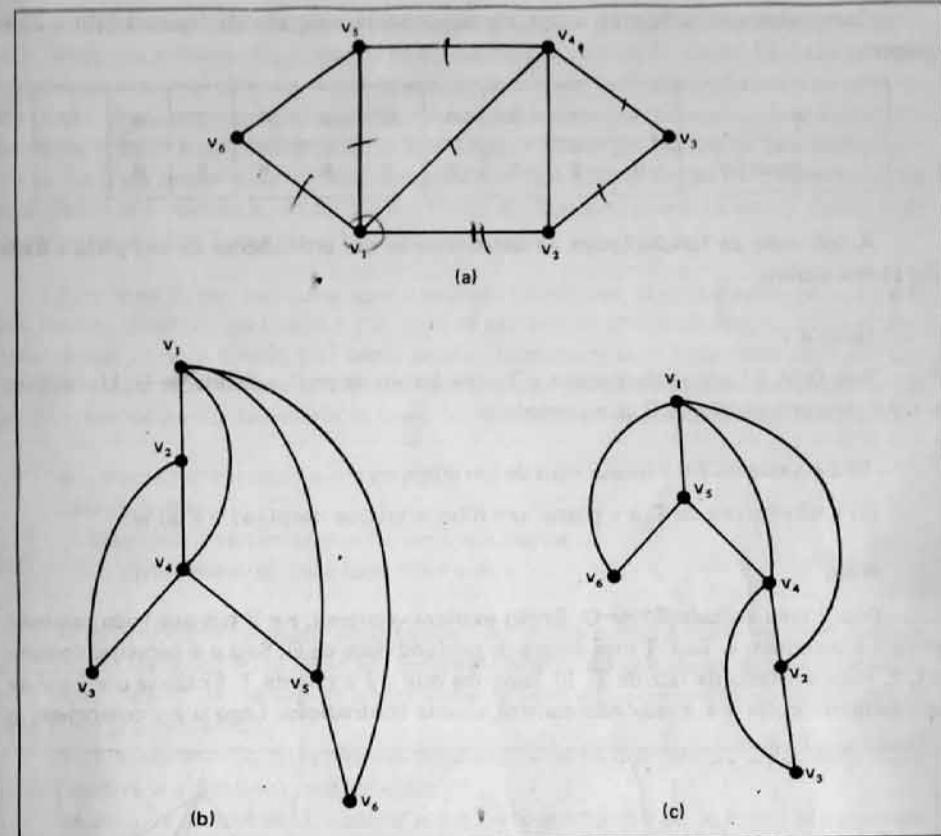


Figura 4.4. Busca em profundidade

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 aos seus componentes conexos, separadamente.

4.4 – Biconectividade

Como aplicação de busca em profundidade, será descrito um algoritmo para a determinação dos componentes biconexos 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 $lowpt : V \rightarrow V$, do seguinte modo. Para cada vértice $v \in V$, $lowpt(v)$ 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 lowpt correspondente ao grafo da figura 4.5(b) é a seguinte:

v	1	2	3	4	5	6	7	8	9	10
lowpt(v)	8	8	2	2	2	8	8	8	8	1

A aplicação da função lowpt na determinação das articulações de um grafo é dada pelo lema abaixo.

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
- (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, 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

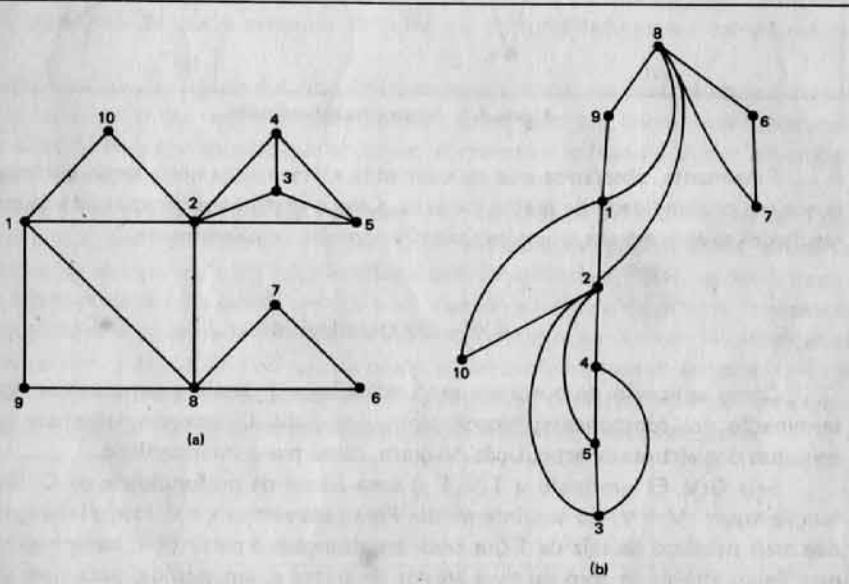


Figura 4.5. Outra busca em profundidade

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 . Então t e z são tais que um deles é ancestral de v e o outro descendente de v em T . Então v não pode ser uma folha. Além disso, porque v é articulação, não pode haver aresta (fronde) unindo algum descendente próprio a 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 acima, 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 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

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

A determinação de 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 os componentes biconexos 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 lowpt .

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 um componente biconexo 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 dos componentes biconexos 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 um componente biconexo em G. Retirar T_w de T. Além disso, se w é o único demarcador de v em T, dourante 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 um componente biconexo. 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 um componente biconexo 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 um componente biconexo 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 os componentes obtidos.

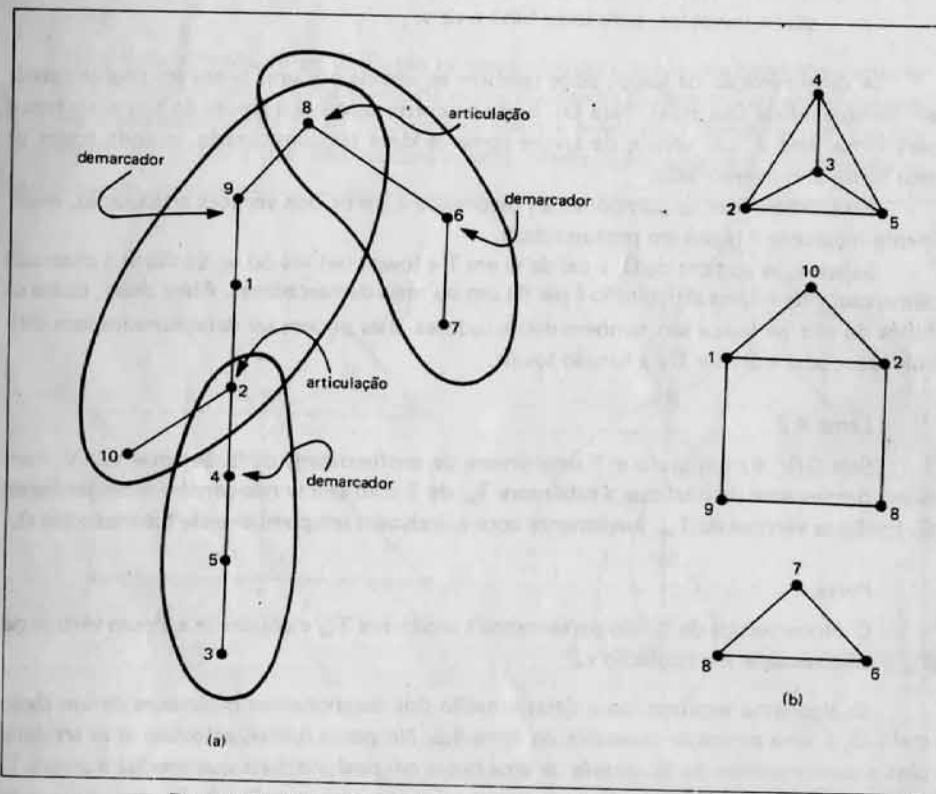


Figura 4.6. Determinação dos componentes biconexos de um grafo

4.5 – Busca em Profundidade – Dígrafos

A idéia 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 acima, 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 seguinte algoritmo recursivo implementa a idéia.

algoritmo 4.3: Busca em profundidade em dígrafos

dados dígrafo $D(V, E)$

procedimento $P(v)$

```

    marcar v
    colocar v na pilha Q
    para w ∈ 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 ∈ V
    P(s)
  
```

Observa-se que a pilha Q (a qual corresponde ao caminho Q da descrição dada) pode ser ignorada no algoritmo 4.3, sem modificar a essência do procedimento. Contudo ela foi incluída para reforçar a idéia 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 dígrafo. 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) denominase *aresta de retorno*, caso contrário ($w \notin Q$), a aresta (v, w) é denominada *aresta de cruzamento*. Desse modo, a busca em profundidade em um dígrafo divide o seu conjunto de arestas em quatro subconjuntos disjuntos.

Seja $D(V, E)$ um dígrafo 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 dígrafo D . Essa árvore é denominada *árvore de profundidade de raiz s* , do dígrafo D .

Teorema 4.4

Seja $D(V, E)$ um dígrafo 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 dígrafo 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 dígrafos possui também complexidade $O(n + m)$.

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

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

Assim como no caso não direcionado, a ordem em que os vértices de um dígrafo $D(V, E)$ 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 , $PE(v)$, e *profundidade de saída* de v , $PS(v)$, 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 acima, seriam as seguintes as profundidades dos vértices do dígrafo da figura 4.7(a).

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

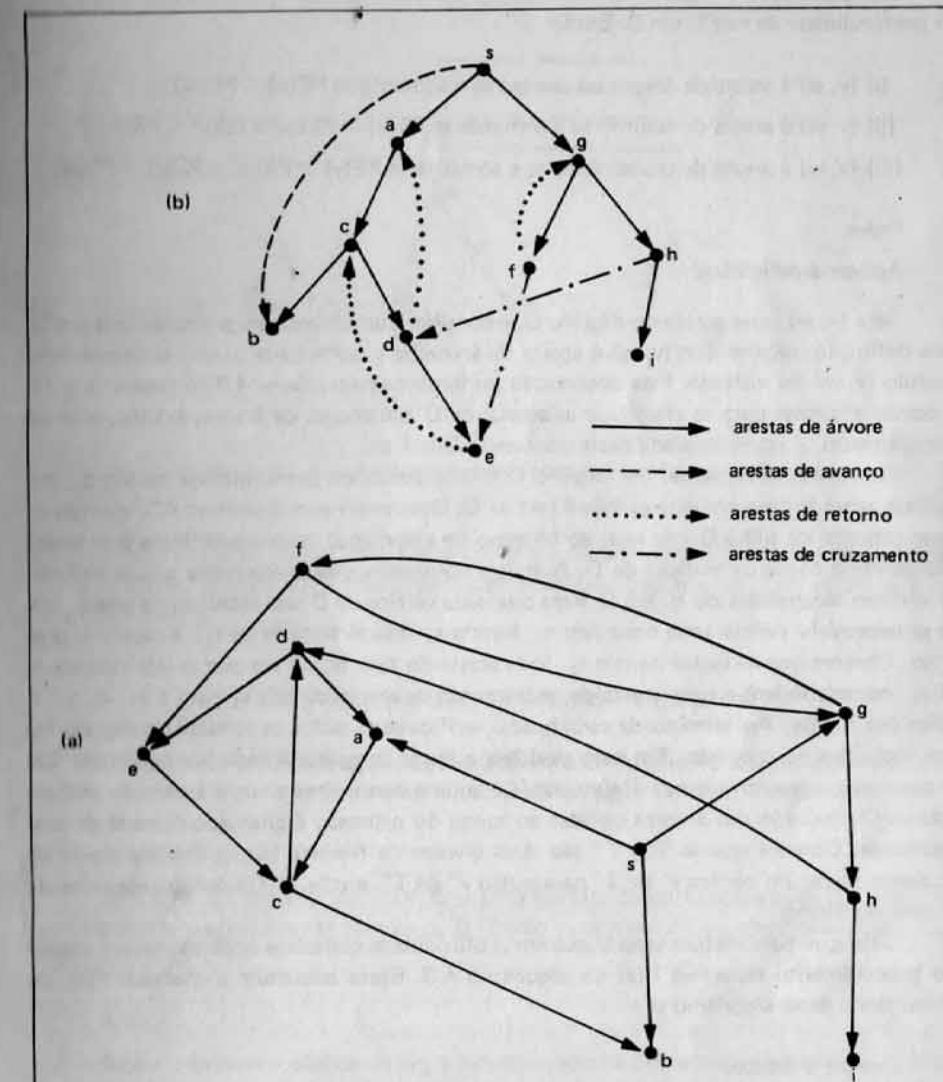


Figura 4.7. Busca em profundidade em dígrafos

Observa-se que a seqüência dos vértices v de um dígrafo D em ordem crescente de $PE(v)$ 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 dígrafo através de uma busca em profundidade, utilizando-se o seguinte lema.

Lema 4.3

Seja $D(V, E)$ um dígrafo 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 $PE(v) < PE(w)$
- (ii) (v, w) é aresta de retorno se e somente se $PE(v) > PE(w)$ e $PS(v) < PS(w)$
- (iii) (v, w) é aresta de cruzamento se e somente se $PE(v) > PE(w)$ e $PS(v) > PS(w)$

Prova

Aplicar a definição.

Seja (v, w) uma aresta do dígrafo 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 dígrafo 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 4.3, 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 (sendo 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 dígrafo 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 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 4.3. Basta substituir a chamada $P(s)$, da última linha desse algoritmo por:

para $s \in V$ efetuar
se s é não marcado então $P(s)$

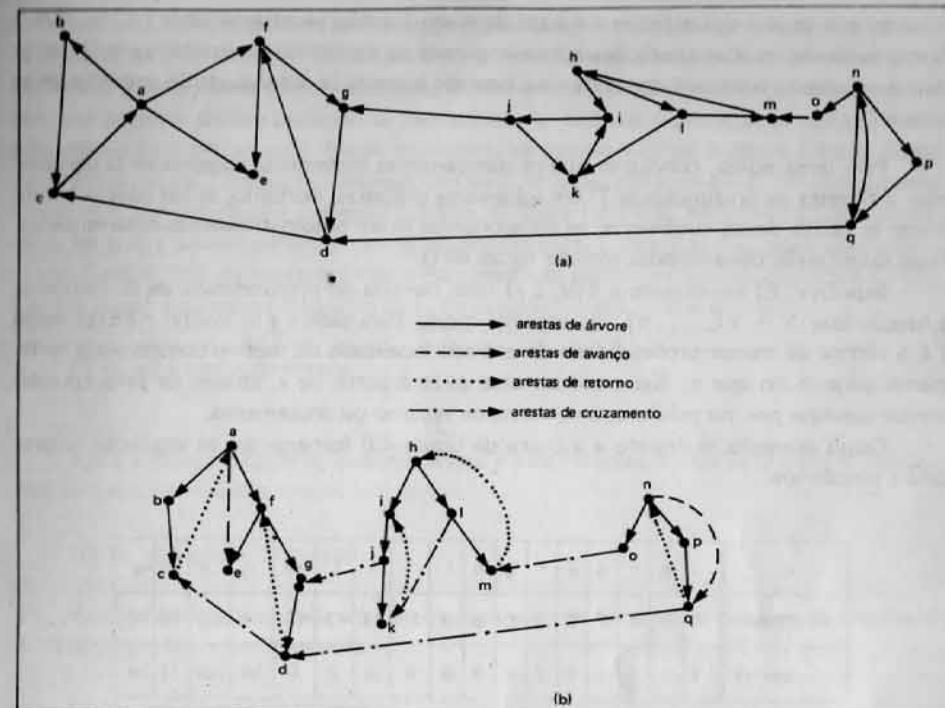


Figura 4.8. Busca em profundidade completa em dígrafos. Caso geral

A figura 4.8(b) ilustra uma busca completa efetuada no dígrafo da 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 Conexos

Nesta seção apresenta-se um algoritmo para a determinação dos componentes fortemente conexos de um dígrafo, o qual ilustra uma aplicação de busca em profundidade.

Lema 4.4

Seja $D(V, E)$ um dígrafo, $T(V, E_T)$ uma floresta de profundidade de D e $S(V_S, E_S)$ um componente fortemente conexo 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 acima implicam na 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 acima, conclui-se que os componentes fortemente conexos de D participam a floresta de profundidade T' em subárvore disjuntas. Portanto, se for possível identificar as raízes dessas subárvore, os componentes ficam automaticamente determinados. Essas raízes serão denominadas *vértices fortes* de D .

Seja $D(V, E)$ um dígrafo 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 no mesmo componente fortemente conexo do que v , que pode ser alcançado a partir de v , através de zero ou mais arestas seguidas por, no máximo, uma aresta de retorno ou cruzamento.

Como exemplo, o dígrafo 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
$\text{PE}(v)$	1	2	3	7	5	4	6	8	9	10	11	12	13	14	15	16	17
$\text{low}(v)$	1	1	1	3	5	3	3	8	9	9	9	8	8	14	15	14	14

A aplicação da função low na determinação dos componentes fortemente conexos se dá através do lema abaixo.

Lema 4.5

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

Prova

Obviamente, $\text{low}(v) \leq \text{PE}(v)$, 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, $\text{low}(v) = \text{PE}(v)$. Reciprocamente, se $\text{low}(v) = \text{PE}(v)$ 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 dos componentes fortemente conexos de um dígrafo, aparentemente apresenta uma circularidade. Para se obter os 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 acima é de fato apenas aparente. O cálculo da função low pode ser realizado simultaneamente à determinação dos componentes fortemente

conexos 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 $\text{low}(v) := \text{PE}(v)$. Este valor será atualizado no decorrer da computação, se necessário. Ao final da exploração de v , $\text{low}(v)$ terá sido calculado corretamente. Neste momento, se $\text{low}(v) = \text{PE}(v)$ então v é forte. Nesse caso retiram-se de T todos os descendentes de v , os quais induzem em D um componente fortemente conexo de raiz v . Caso contrário, se $\text{low}(v) \neq \text{PE}(v)$ 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 $\text{low}(v)$, 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 $\text{low}(w)$. Então se $\text{low}(w) < \text{low}(v)$, deve ser feita a atribuição $\text{low}(v) := \text{low}(w)$.

(ii) (v, w) é aresta de avanço:

As arestas de avanço não alteram os componentes fortemente conexos de um dígrafo. Podem, portanto, ser ignoradas.

(iii) (v, w) é aresta de retorno:

Nesse caso v, w pertencem necessariamente ao mesmo componente fortemente conexo. Então, se $\text{PE}(w) < \text{low}(v)$ efetua-se $\text{low}(v) := \text{PE}(w)$.

(iv) (v, w) é aresta de cruzamento:

Surge o problema de determinar se v pertence ou não ao componente fortemente conexo 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. Conseqüentemente 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 $\text{PE}(w) < \text{low}(v)$, efetua-se $\text{low}(v) := \text{PE}(w)$.

Essas observações conduzem à seguinte formulação.

algoritmo 4.4: Componentes fortemente conexos de um dígrafo

dados dígrafo $D(V, E)$

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

$F(w)$

$low(v) := \min \{ low(v), low(w) \}$

caso contrário

se w está em T então $low(v) := \min \{ low(v), PE(w) \}$

se $low(v) = PE(v)$ então retirar v e seus descendentes de T

(induzem um componente fortemente conexo)

$j := 0$

desmarcar todos os vértices

definir uma árvore T , vazia

para $s \in V$ efetuar se s não marcado então $F(s)$

O algoritmo acima pode ser implementado sem dificuldade em tempo $O(n + m)$. Como exemplo, na busca da figura 4.8(b) correspondente ao dígrafo 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 o dá origem a um componente constituído apenas por ele mesmo e o vértice n corresponde ao componente $\{n, p, q\}$. A figura 4.9 ilustra a saída produzida.

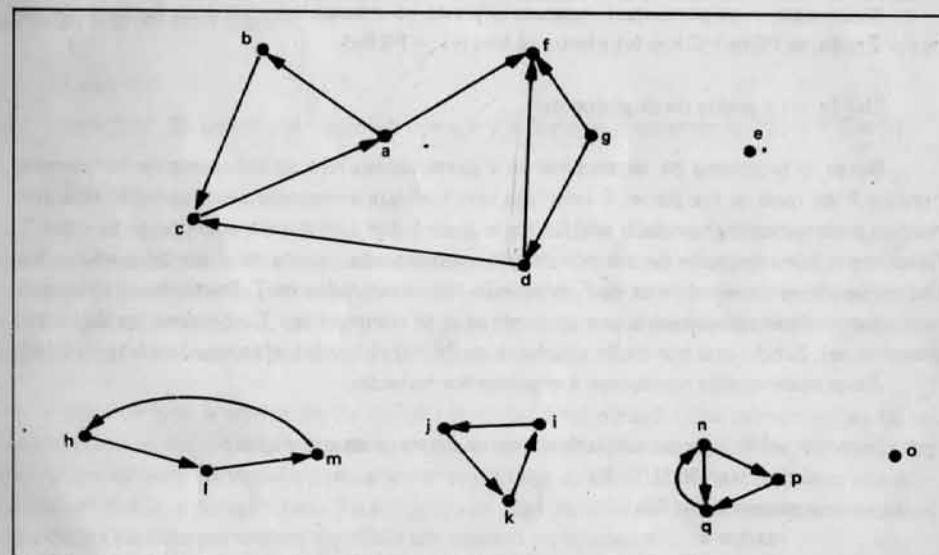


Figura 4.9. Componentes fortemente conexos do dígrafo da figura 4.8(a)

4.7 – Busca em Largura

No capítulo 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 obedece a:

"dentre 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.

algoritmo 4.5: Busca em largura

dados grafo $G(V, E)$, conexo

desmarcar todos os vértices

escolher uma raiz $s \in V$

definir uma fila Q , vazia

marcar 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) (I)

marcar w

inserir w em Q

caso contrário

se $w \in Q$ então visitar (v, w) (II)

retirar v de Q

À semelhança da busca em profundidade, o critério de escolha de arestas também é arbitrário nesse caso. Também no algoritmo 4.5 esse critério é dado pela ordenação de $A(v)$.

As visitas a arestas são realizadas nas linhas (I) e (II) acima. 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 . Normalmente T será considerada como árvore enraizada, sendo sua raiz precisamente o vértice raiz da busca.

Para cada $v \in V$, seja $nível(v)$ o nível do vértice v na árvore T .

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 $nível(v) \neq nível(w)$ diferem, no máximo, de uma unidade.

Prova

Seja v alcançado antes de w na busca em largura correspondente à árvore T . Então $nível(v) \leq nível(w)$. Agora se $nível(v) \neq nível(w)$ 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 4.5, 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. Arestras (v, w) tais que $|nível(v) - nível(w)| = 1$ e arestras em que $nível(v) - nível(w) = 0$. Cada uma dessas partes pode ser ainda particionada em duas outras. Seja a aresta (v, w) com $nível(v) \leq nível(w)$:

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

A figura 4.10(b) ilustra 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(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) obedecem 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
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 um de seus componentes conexos.

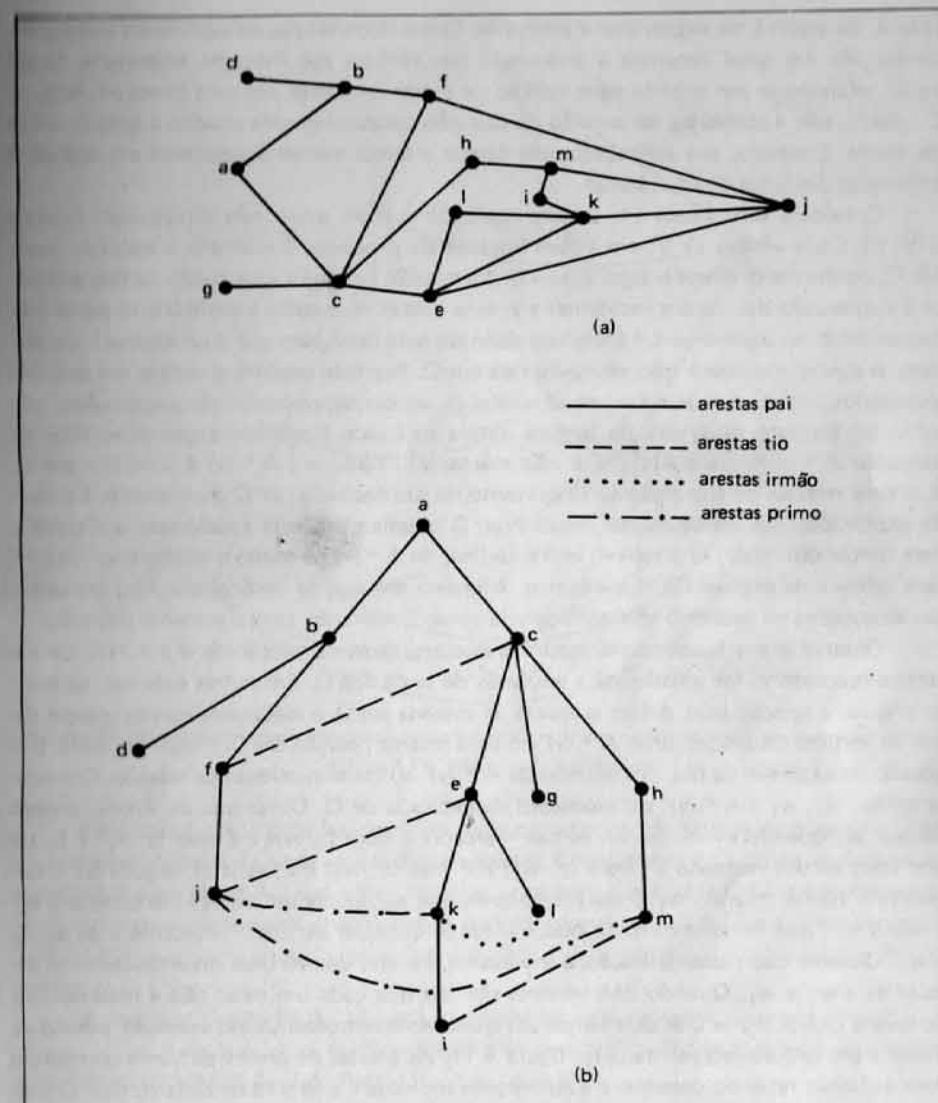


Figura 4.10. Busca em largura

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 4.5. 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 4.4 é o bloco definido pela linha *para* $w \in A(v)$ efetuar). 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 influi no formato da árvore de largura obtida na busca. Considere agora os vértices do conjunto $A^*(v) = \{w \in A(v) \mid w \text{ é não marcado}\}$. 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 idéia 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. Considerando 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 $(z', w_1) \in E$ 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 $A^*(7) = \{11, 12, 13\}$, 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 seqüência 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 seqüência em que os vértices são retirados da fila utilizada na busca.

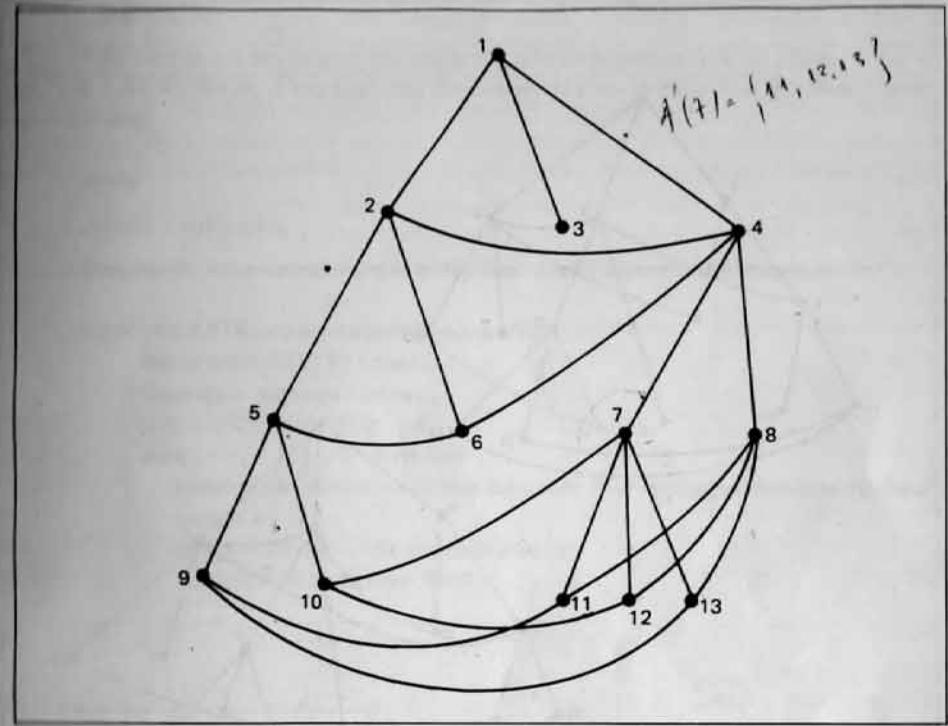


Figura 4.11. Uma busca em largura

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 seqüências de inteiros. Diz-se que S_1 é *lexicograficamente maior* do que S_2 , denotando $S_1 > S_2$, quando:

- (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 seqüências 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 $A < B < \dots < Z$.

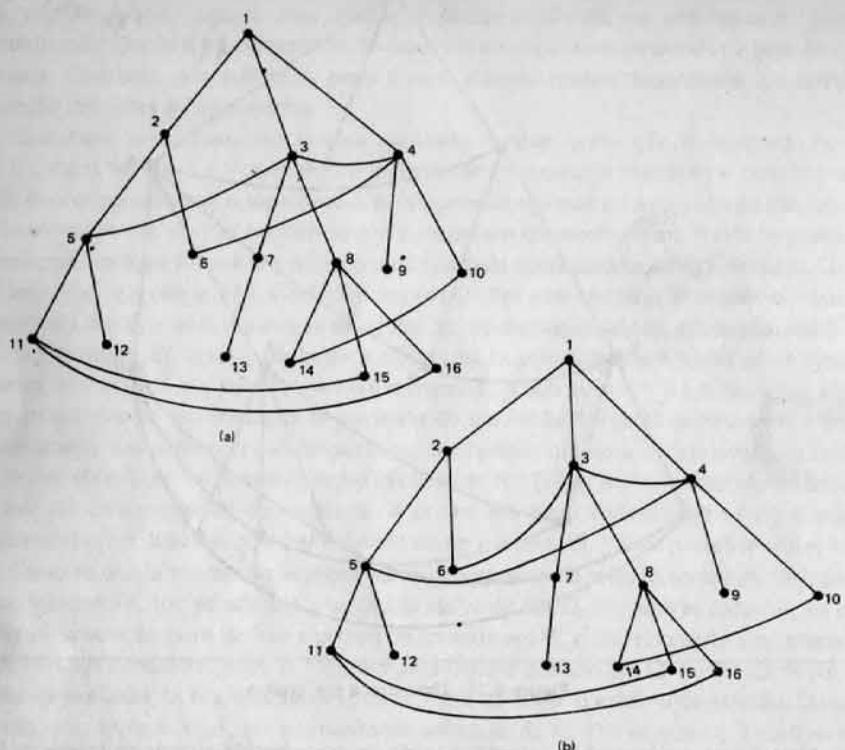


Figura 4.12. Busca em largura lexicográfica e não lexicográfica

Seja uma busca em largura em um grafo não direcionado $G(V, E)$. A idéia consiste em utilizar uma seqüência 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 seqüência dos complementos das larguras dos z_i . 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 seqüência 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 acima, pode-se então descrever o algoritmo correspondente.

algoritmo 4.6: 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 máximo

marcar v

para $w \in A(v)$ e w não marcado efetuar

incluir j à direita em $R(w)$

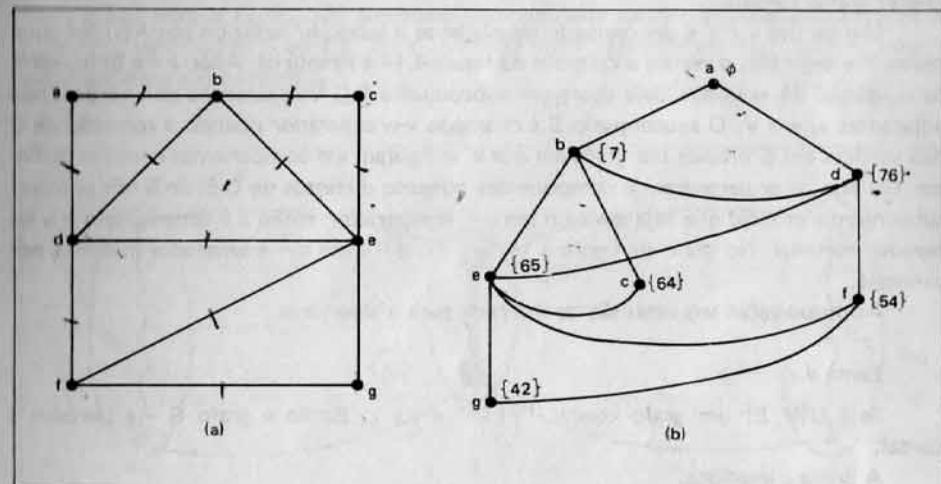


Figura 4.13. Busca em largura lexicográfica com rotulação de vértices

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 4.13(a). A seqüência de rótulos $R(w)$ está indicada para cada caso.

A rotulação R efetuada pelo algoritmo 4.6 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 $O(n + m)$.

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 suponha 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 conexos distintos. Ou seja, v, w pertencem a componentes conexos distintos 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

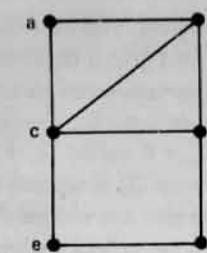


Figura 4.14. Grafo não cordal

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 os componentes conexos 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 ≥ 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.

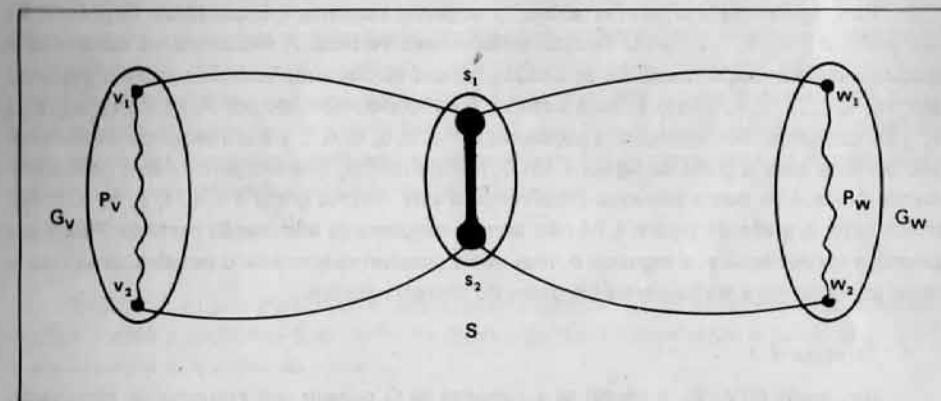


Figura 4.15. Prova do teorema 4.7

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 os componentes conexos 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 haja 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 acima possa ser aplicado, restam ainda duas tarefas importantes a serem realizadas:

- Provar que se o processo acima esgotar todos os vértices de G , então o grafo é necessariamente cordal.
- Encontrar uma forma eficiente de localizar vértices simpliciais.

Para realização das tarefas acima, 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 seqüência α 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 seqüência $\alpha = a, c, b, d, e, f, g$ é um esquema de eliminação perfeita para o grafo da figura 4.13(a). Naturalmente, essa seqüência 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 abaixo.

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 acima descrito 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 es-

quema. 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 .

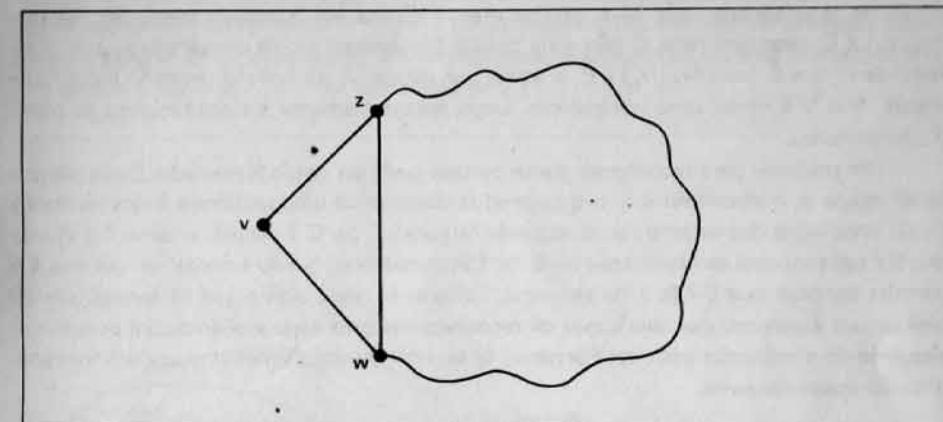


Figura 4.16. Prova do teorema 4.9

O teorema 4.9 responde à questão (i) acima colocada. 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 seqüência 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 4.6 de busca em largura lexicográfica. Então a seqüência S de vértices v ordenados decrescentemente segundo largura(v) é um esquema de eliminação perfeita.

Prova

Por definição de largura(v), a seqüência S corresponde à ordem inversa em que os vértices de G são retirados da fila pelo algoritmo 4.6. 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. Então, novamente 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. Então, $(v_4, v_5) \in E$, 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 4.6, 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.

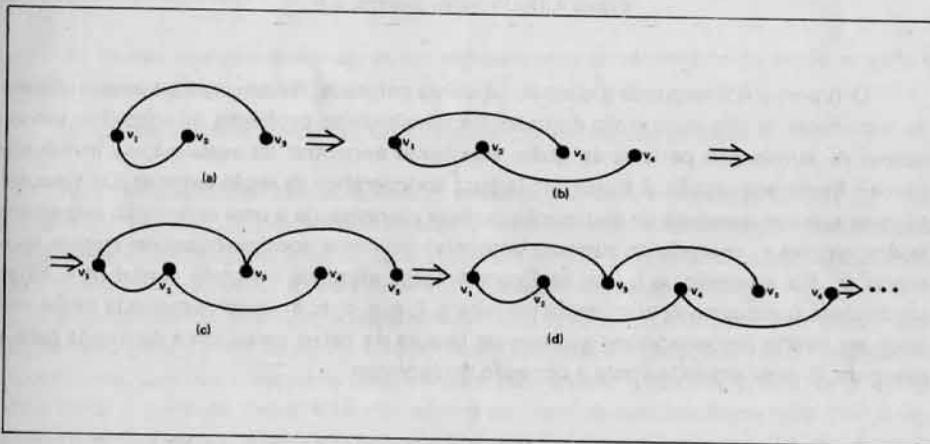


Figura 4.17. Prova do lema 4.10

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 largura(v), obtida como saída de uma busca em largura lexi-

gráfica. Para cada vértice $v_j \in V$, define-se o multiconjunto $L(v_j)$, inicialmente vazio. A idéia 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 . S é um esquema de eliminação perfeita se e somente se a igualdade $L(v_j) - A(v_j) = \emptyset$ for verdadeira para $j = 1, \dots, n$.

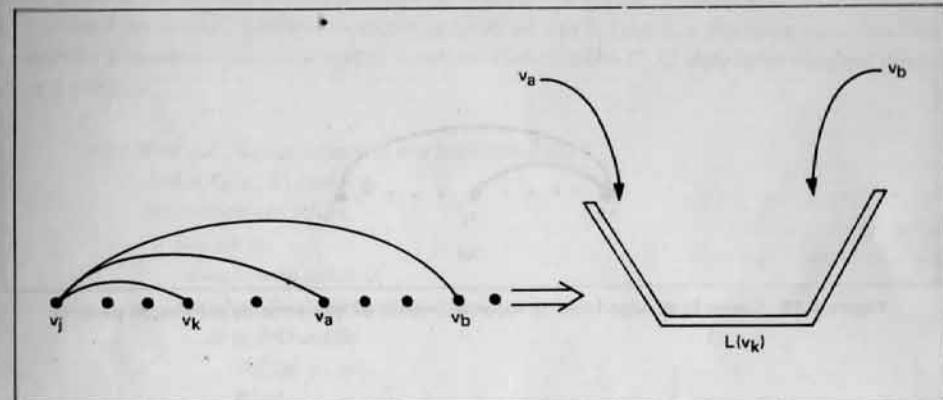


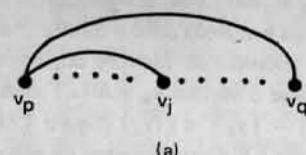
Figura 4.18. Passo do algoritmo de reconhecimento de esquemas de eliminação perfeita

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

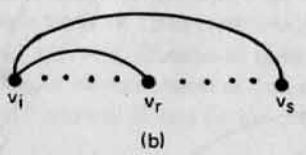
$$\begin{array}{ll} L(g) = \emptyset & L(e) = \{d, b\} \\ L(f) = \{e\} & L(d) = \{b\} \\ L(c) = \emptyset & L(b) = \{a\} \\ & L(a) = \emptyset \end{array}$$

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) $(v_j, v_q) \in E$. 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. Consequentemente, S não é um esquema de eliminação perfeita. Reciprocamente, seja a suposição de que $L(v_j) - A(v_j) = \emptyset$, 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 $(v_r, v_s) \in E$ (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 $(v_r, v_s) \in E$.



(a)



(b)

Figura 4.19. Correção do algoritmo de reconhecimento de esquemas de eliminação perfeita

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. Aplica-se uma busca em largura lexicográfica e obtém-se uma seqüência 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 conceitual 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 idéia 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 acima 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 no 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.

algoritmo 4.7: Busca irrestrita em profundidade

```

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  $s$ 
     $P(s)$ 
```

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, enquanto permanecer na pilha. A ocorrência de chamada recursiva $P(w)$, com w pertencendo à pilha Q , pode causar 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 4.7. 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) $i \neq j \Rightarrow C_i \neq C_j$.

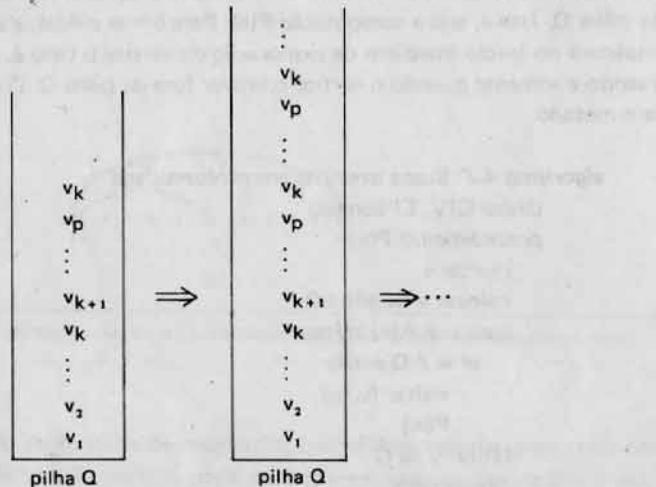


Figura 4.20. Ciclicidade

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 $P(v)$ do algoritmo 4.7, 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 4.7. 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.

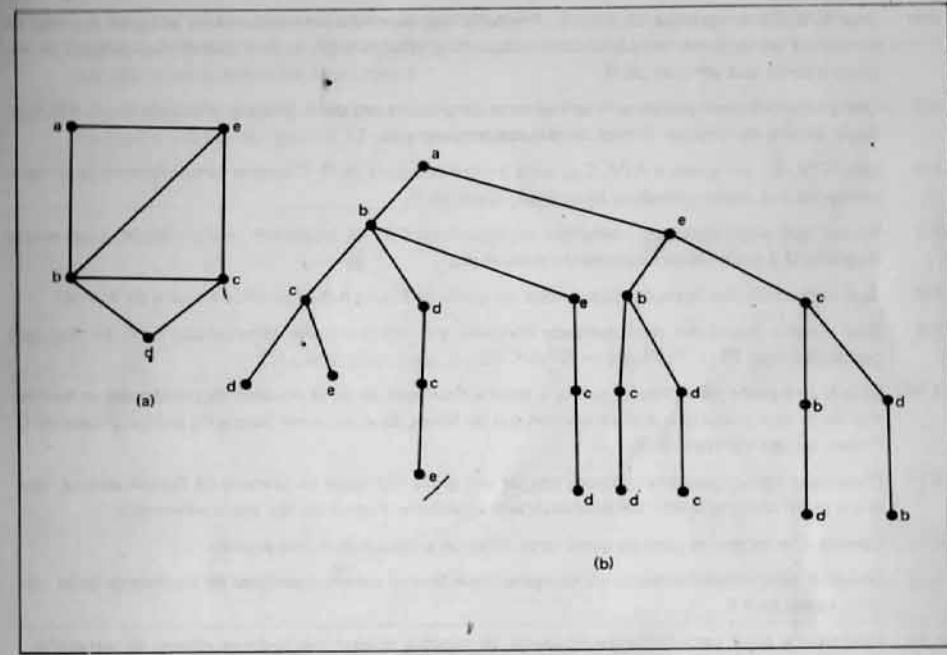


Figura 4.21. Busca irrestrita em profundidade de raiz a

Uma consequência do lema acima é que o algoritmo 4.7, na realidade, determina todos os caminhos maximais de G.

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 marcado quando é incluído na pilha Q. Ao ser excluído de Q, q 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.

4.11 – EXERCÍCIOS

- 4.1 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 4.4(b)?
- 4.9 Seja B uma busca em profundidade efetuada em um grafo não direcionado $G(V, E)$. Em que condições vale: $PE(v) < PE(w) \Rightarrow PS(v) < PS(w)$, para todo $v, w \in V$?
- 4.10 Seja G um grafo não direcionado e v uma articulação de G. O número de componentes biconexos 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 biconexos 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 Detalhar uma implementação de complexidade linear, para o algoritmo de biconectividade, descrito na seção 4.4.
- 4.14 Modificar o algoritmo de biconectividade, de modo a determinar todas as pontes de um grafo.
- 4.15 Formular uma descrição do algoritmo 4.3, de busca em profundidade em dígrafos, de modo a eliminar a pilha Q.
- 4.16 Seja $D(V, E)$ um dígrafo, $(v, w) \in E$ e B uma busca em profundidade em D. Provar que se (v, w) é aresta de avanço então $PE(v) < PE(w) + 1$. Vale a recíproca?
- 4.17 Seja D um dígrafo dado. Elaborar um algoritmo para reconhecer se uma dada árvore direcionada enraizada T é árvore de profundidade de D.
- 4.18 Um dígrafo D é acíclico se e somente se existir uma busca em profundidade em D sem arestas de retorno. Provar ou dar contra-exemplo.
- 4.19 Caracterizar os dígrafos $D(V, E)$ com raiz s e V, para os quais toda busca em profundidade com raiz s não produz arestas de cruzamento.
- 4.20 Seja $D(V, E)$ um dígrafo. 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.
- 4.21 Seja D um dígrafo 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.

- 4.22 Seja $D(V, E)$ um dígrafo e $(v, w) \in E$ tal que (i) v, w pertencem a componentes fortemente conexos distintos de D e (ii) $PE(w) < PE(v)$ em uma busca em profundidade efetuada em D. Então no algoritmo 4.3, w ∈ Q no momento da visita à aresta (v, w) . Provar ou dar contra-exemplo.
- 4.23 Sobre a computação da função *low*, conforme descrita na seção 4.6. Provar que a mesma está correta.
- 4.24 Descrever outra formulação do algoritmo 4.4, empregando uma pilha em lugar de árvore T.
- 4.25 Um dígrafo 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.
- 4.26 Provar que um dígrafo D é estruturado em árvore se e somente se seu fechamento transitivo não contiver o dígrafo da figura 4.22 como subgrafo induzido.

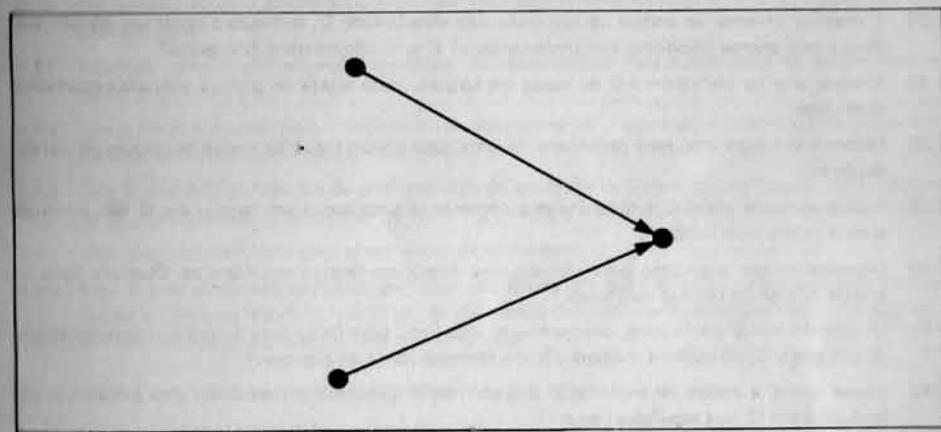


Figura 4.22. O subgrafo proibido para o fechamento transitivo de dígrafos estruturados em árvore

- 4.27 Seja uma busca em profundidade realizada num grafo não direcionado e conexo G. Seja D o dígrafo 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.
- 4.28 Descrever um algoritmo de complexidade linear para reconhecer dígrafos estruturados em árvore.
- 4.29 Descrever um algoritmo de complexidade linear para determinar a redução transitiva de um dígrafo estruturado em árvore.
- 4.30 Descrever um algoritmo para determinar o fechamento transitivo de um dígrafo estruturado em árvore, tal que possua complexidade linear no tamanho de sua saída.
- 4.31 Um dígrafo *série paralelo minimal* (SPM) é definido recursivamente por:
 - (i) um dígrafo trivial (com um único vértice) é SPM e
 - (ii) se $D_1(V_1, E_1)$ e $D_2(V_2, E_2)$ são dígrafos SPM então também o são os dígrafos $(V_1 \cup V_2, E_1 \cup E_2)$ e $(V_1 \cup V_2, E_1 \cup E_2 \cup (A_1 \times B_2))$, onde A_1, B_2 são os conjuntos das fontes de D_1 e sumidouros de D_2 , respectivamente.

Dê exemplo de um dígrafo que seja estruturado em árvore, mas não SPM. Em seguida, exemplifique um outro que seja SPM, mas não estruturado em árvore.

- 4.32 Um dígrafo acíclico é *série paralelo geral* (SPG) quando a sua redução transitiva for SPM. Mostrar que todo dígrafo estruturado em árvore é SPG.
- 4.33 Elaborar um algoritmo para reconhecer se num dado dígrafo 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 dado k (a que corresponde o caso $k = 0$?).
- 4.34 Mostrar que as arestas de todo grafo não direcionado G podem ser orientadas de tal forma que no dígrafo resultante não existam arestas implícitas por transitividade. Desenvolver um algoritmo de complexidade linear para efetuar esta orientação.
- 4.35 Mostrar que as arestas de todo grafo não direcionado G podem ser orientadas de tal forma que o dígrafo resultante seja acíclico. Apresentar o algoritmo correspondente, de complexidade linear.
- 4.36 É possível orientar as arestas de um grafo não direcionado G , de modo a obter um dígrafo acíclico e sem arestas implícitas por transitividade? E se G não contiver triângulos?
- 4.37 Mostrar que no algoritmo 4.5 de busca em largura, cada aresta do grafo é visitada exatamente duas vezes.
- 4.38 Elaborar um algoritmo para reconhecer se uma dada árvore é ou não árvore de largura de um dado grafo.
- 4.39 Provar que um grafo G é bipartite se e somente se uma busca em largura em G não produzir arestas primo nem irmão.
- 4.40 Desenvolver um algoritmo para efetuar uma busca em largura em dígrafos. Quantos tipos de arestas diferentes produz essa busca?
- 4.41 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?
- 4.42 Numa busca, a ordem de exploração dos vértices corresponde à ordem em que os mesmos são incluídos em Q , nos seguintes casos:
- (i) a busca é de profundidade e Q é a pilha associada à busca.
 - (ii) a busca é de largura e Q é a fila associada à busca.
- Provar ou dar contra-exemplo.
- 4.43 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 4.6), escolher sempre a aresta menos remota. Provar ou dar contra-exemplo.
- 4.44 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?
- 4.45 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 $|V|$.
- 4.46 Mostrar que a condição “escolher um vértice v marcado com $R(v)$ lexicograficamente máximo”, no algoritmo 4.6 de busca em largura lexicográfica, satisfaz à escolha de v através da utilização de uma fila.
- 4.47 Modificar o algoritmo 4.6 de busca em largura lexicográfica, de modo a obter explicitamente a árvore de largura correspondente.

4.48 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.

4.49 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.

4.50 Reconhecer se uma árvore dada é ou não árvore de largura lexicográfica de um grafo não direcionado dado.

4.51 Formular uma implementação detalhada do algoritmo de reconhecimento de grafos cordais, descrito na seção 4.9.

4.52 Um grafo é *4-cordial* quando todo ciclo de comprimento > 4 possui uma corda. Elaborar um algoritmo para reconhecer grafos 4-cordais.

4.53 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.

4.54 Descrever um algoritmo para busca irrestrita em largura.

4.55 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.

4.56 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 $L(S)/2$ 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.

4.57 Provar o lema 4.9.

4.58 O seguinte algoritmo se destina a enumerar os ciclos simples de um dígrafo.

```

algoritmo enumeração de ciclos
    dados dígrafo D(V, E)
    procedimento C(v)
        inserir v em Q
        para w ∈ A(v) efetuar
            se w ∈ Q então C(w)
            caso contrário seja w, ..., v a seqüência de vértices em Q, de w para o topo
                listar ciclo w, ..., v, w
            retirar v de Q
        definir pilha Q
        escolher s ∈ V
        C(s)▲

```

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

4.13 – NOTAS BIBLIOGRÁFICAS

As técnicas de busca, inclusive em profundidade, já são conhecidas desde o século passado, 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). 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 passada. 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 dos componentes biconexos de um grafo (seção 4.4) e dos componentes fortemente conexos de um dígrafo (4.6). O teorema 4.3 aparece em Szwarcfiter, Persiano e Oliveira (1981). Dentre as inúmeras aplicações de busca que se seguiram ao mencionado artigo de Tarjan, as seguintes podem ser mencionadas: determinação dos componentes triconexos de um grafo (Hopcroft e Tarjan (1973)), reconhecimento de grafos planares (Hopcroft e Tarjan (1974)), enumeração dos ciclos simples de um dígrafo (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 dígrafo. A determinação de dominadores em um dígrafo pode ser encontrada em Tarjan (1974) e o reconhecimento de dígrafos reduzíveis 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 idéia 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 o 4.9 de Fulkerson e Gross (1965). Várias provas apresentadas na seção 4.9 foram elaboradas a partir de Golumbic (1980). Os dígrafos estruturados em árvore, exercícios 4.25 e 4.32, podem ser encontrados em Szwarcfiter (1980). Os dígrafos série paralelo, exercícios 4.31 e 4.32, foram estudados por Valdes (1978) e Valdes, Tarjan e Lawler (1979).

CAPÍTULO 5

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 máximo 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 se 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 α .

Ou seja, S' é o maior (ou menor) subconjunto de S , segundo α , que satisfaz P . O algoritmo guloso para resolver este problema consiste num processo iterativo em que S' é construí-

do, 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 α , 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 α , (minimiza α) 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 acima. 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 seguinte, 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 Máxima

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 a 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 máximo, 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 máximo.

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 se maximizar 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) no conjunto de arestas E_T até então obtido não produz ciclos e (ii) o peso total de $E_T \cup \{(v, w)\}$ é máximo, 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 maximiza o peso de $E_T \cup \{(v, w)\}$ é evidentemente a aresta de maior 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 crescente de peso, rejeitando, contudo, cada uma que formar ciclo com aquelas já incluídas em E_T (isto é, com as não rejeitadas)".

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.

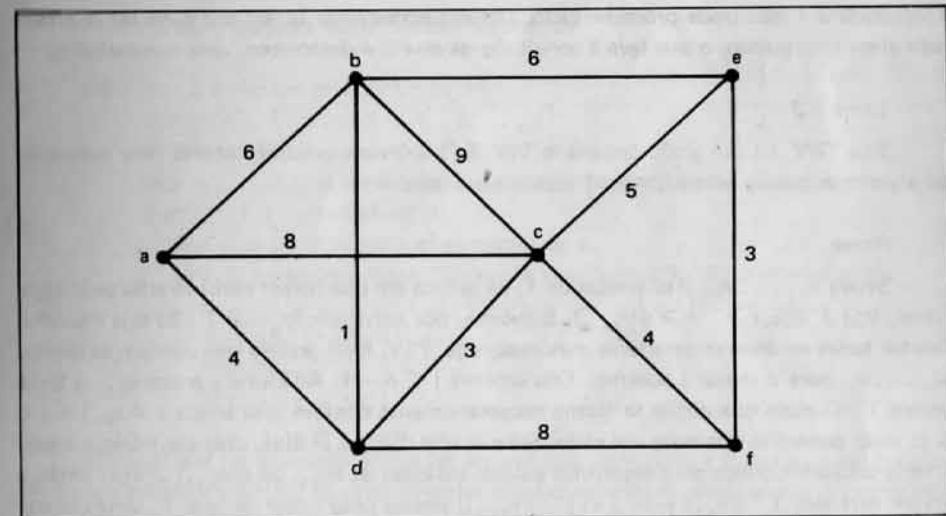


Figura 5.1. Um grafo com pesos nas arestas

As figuras seguintes ilustram o caso. Seja determinar a árvore geradora de maior 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 árvores triviais, com rótulos de {a, b, c,

$d, e, f\}$, respectivamente. Em seguida, as arestas são consideradas em ordem não crescente de pesos. A aresta (b, c) de peso 9 é selecionada em primeiro lugar. Em seguida as arestas (d, f) e (a, c) , de peso 8, numa ordem arbitrária. A aresta (a, b) de peso 6 é rejeitada em seqüência, pois produziria o ciclo a, b, c, a . A aresta (b, e) de peso 6 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 máximo.

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 seqüência ordenada de arestas tal que v está em T_1 e w em T_2 , quando adicionada a T não pode produzir ciclo. Conseqüentemente $\{v, w\}$ 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 máximo.

Prova

Sejam e_1, \dots, e_{n-1} as arestas de T , na ordem em que foram consideradas pelo algoritmo, isto é, $d(e_1) \geq \dots \geq d(e_{n-1})$. Suponha, por contradição, que T não seja máxima. Dentre todas as árvores geradoras máximas, 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 $x \neq e_i$, $1 \leq i \leq p$, caso contrário não seria um ciclo. Sabe-se que $d(e_{j+1}) \geq d(x)$, 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 maior do que T' , uma contradição (figura 5.3). Se $d(e_{j+1}) = d(x)$, então a árvore T^* também é máxima e contém as arestas $e_1, e_2, \dots, e_j, e_{j+1}$ o que contradiz a hipótese de que j era o maior possível. Logo, T deve ser máxima. ▲

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

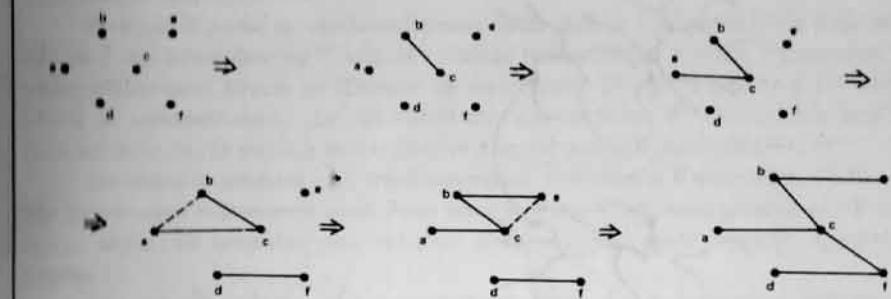


Figura 5.2. Algoritmo de árvore geradora máxima. As linhas tracejadas correspondem às arestas rejeitadas

A formulação seguinte implementa o algoritmo.

algoritmo 5.1: Árvore geradora máxima

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 crescentes.

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.

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 $O(m\log n)$.

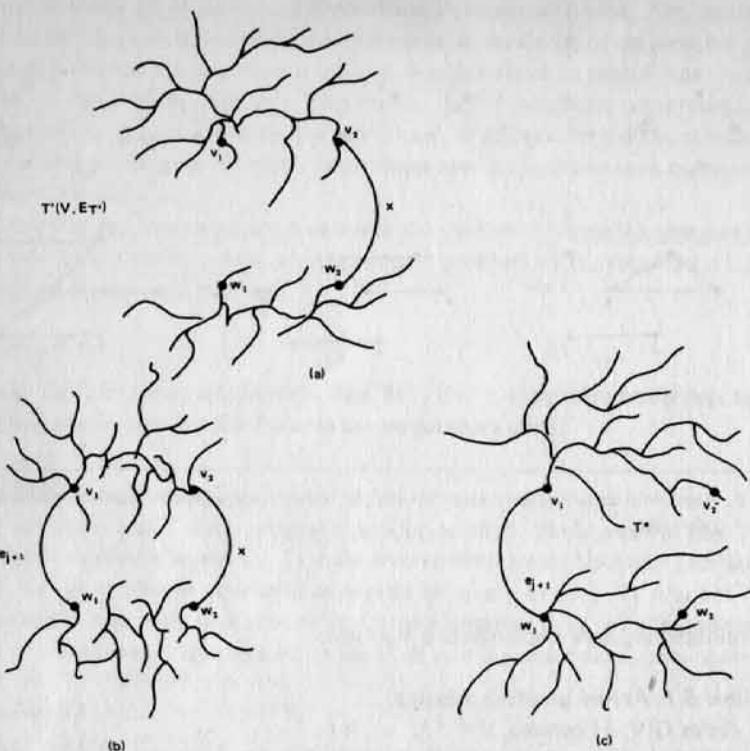


Figura 5.3. Prova do lema 5.2.

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 . Freqüentemente, 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 idéia 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 idéia 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' estejam 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, freqüentemente, 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 *seqüência de Fibonacci*. Esta consiste de uma seqüência de inteiros, cujos primeiros dois elementos são 0 e 1, respectivamente. Subseqüentemente, cada inteiro da seqüência é definido como a soma dos dois imediatamente anteriores. Denotando por $F(0)$ o primeiro elemento da seqüência, por $F(1)$ o seu segundo elemento e 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 se determinar $F(n - 1)$ e $F(n - 2)$, respectivamente. Para se calcular $F(n - 1)$, com $n - 1 > 1$, necessita-se dos resultados dos subproblemas $F(n - 2)$ e $F(n - 3)$. A idéia 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 seqüência que se deseja calcular. A figura 5.4 ilustra o método para $n = 10$. Observa-se que cada elemento da seqüência é calculado em tempo constante, iniciando-se a computação de $n = 0$ e prosseguindo em ordem crescente. Logo, a complexidade do processo é $O(n)$.

n	1	2	3	4	5	6	7	8	9	10
$F(n)$	0	1	1	2	3	5	8	13	21	34

Figura 5.4. Seqüência de Fibonacci, para $n = 10$

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

5.5 – Particionamento de Árvores

Nessa 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 partitionar 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) $\sum_{e \in E_i} d(e)$ mínimo

Observe que o partitionamento de T em subárvores é perfeitamente determinado por um partitionamento $\{V_1, \dots, V_t\}$ de seus vértices, $\cup V_i = V$ 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 partitionam 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 $e \in E - E_i$, $1 \leq i \leq t$, 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, $\sum_{1 \leq j \leq t} d(e)$. 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 partitionamento 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 pois pode ser também 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 partitionamento em subárvores, com $k = 3$. Este corresponde ao seguinte partitionamento de vértices:

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

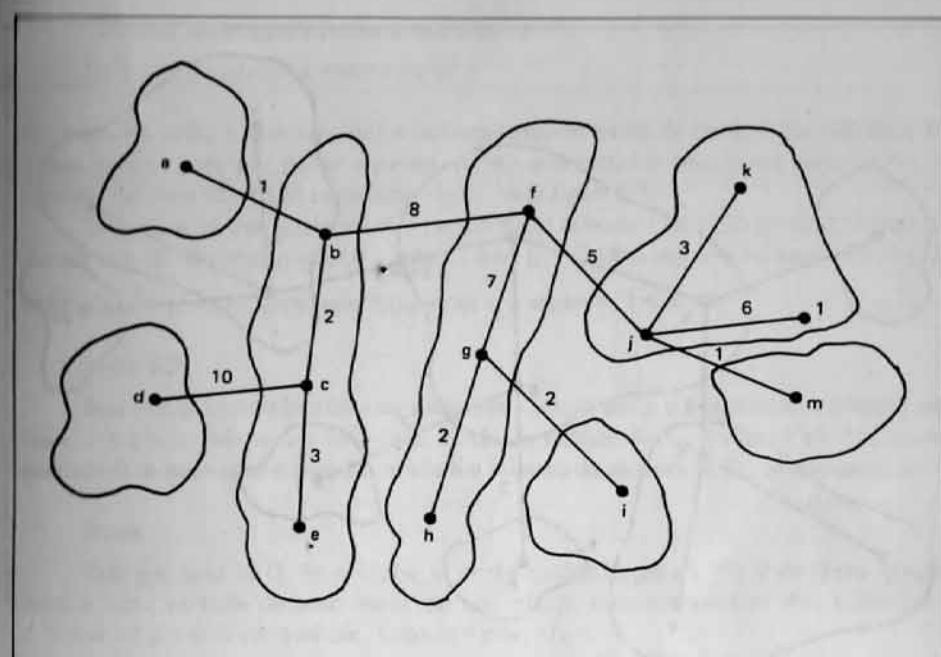


Figura 5.5. Um partitionamento de uma árvore

As arestas de corte da partição indicada são (a, b), (d, c), (b, f), (f, g), (g, i) e (i, 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 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, $1 \leq j \leq k$.

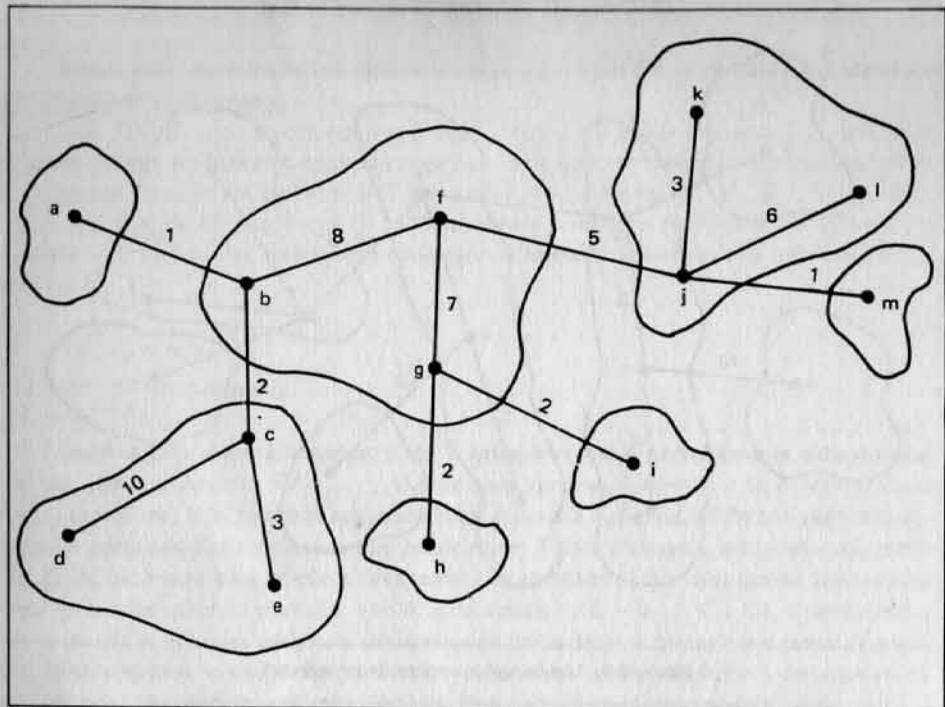


Figura 5.6. Um particionamento ótimo

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 $d(v, w_i)$ deve constituir parcela de $p(v, j)$. Veja figura 5.7.

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 , 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) acima. 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 $P(v, j)$.

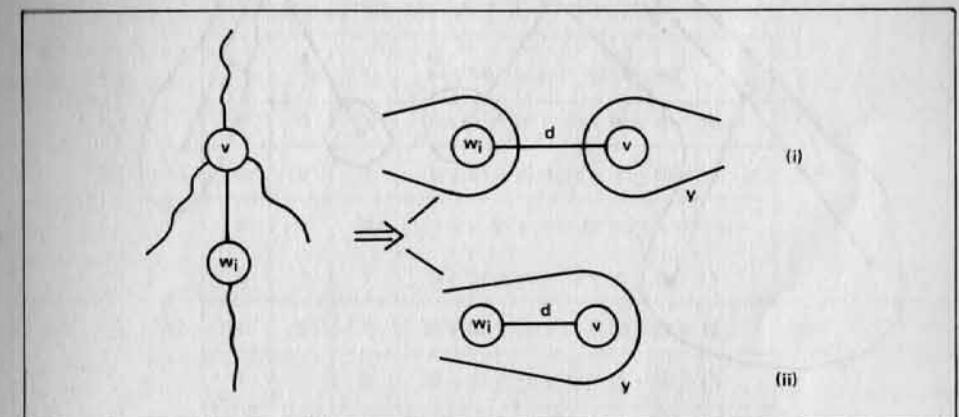


Figura 5.7. Construção de $P(v, j)$

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:

$$\alpha(h) = 1 \text{ se } h \neq 0$$

$$\alpha(h) = 0 \text{ se } h = 0.$$

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

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

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 $h_i \neq 0$.

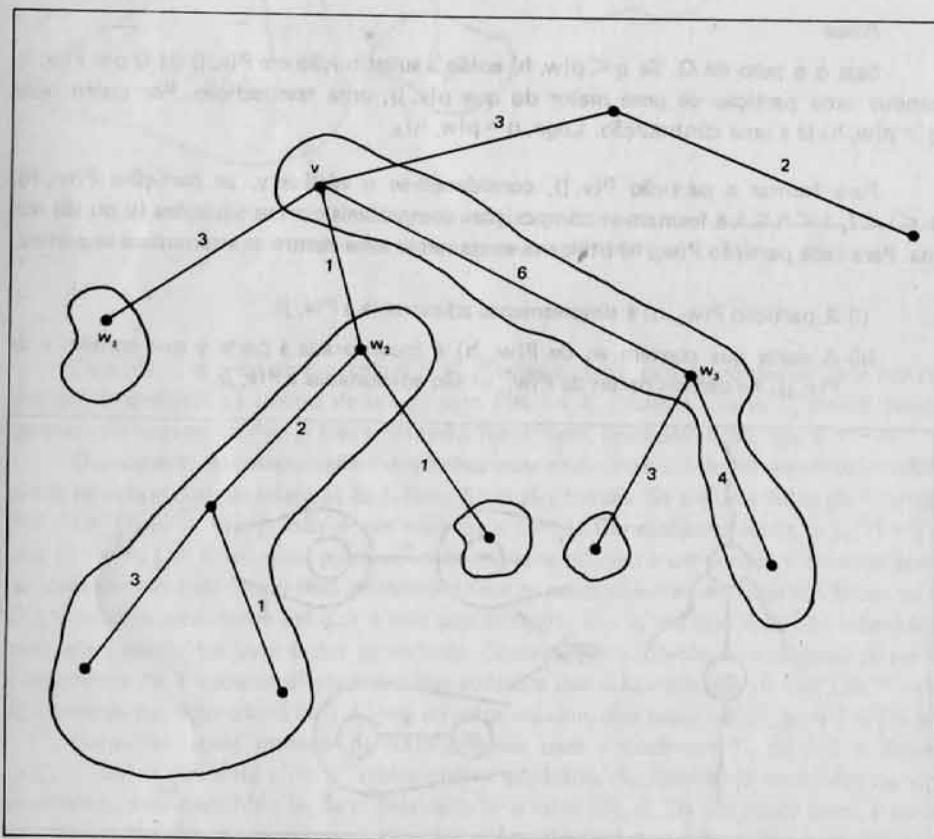


Figura 5.8. Construção de $P(v, 3)$, sendo $k = 4$

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 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.

	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 $p(v, 3)$

A tabela da 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

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

Note que h_i representa o número de elementos de T_{w_i} pertencentes à parte que contém v , da partição candidata a $P(v, 3)$. Naturalmente, $P(v, 3)$ será a de maior peso dentre elas. Examinando 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.

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 $p(v, 3)$

A aplicação do processo acima, evidentemente, resolve o problema. Calcula-se $p(v, j)$, para todo j , $0 \leq j \leq k$ e todo vértice v da árvore T . O peso do particionamento ótimo da árvore será então $p(r, 0)$, onde r é a raiz de T . Contudo, $p(v, j) = \max_{\sum h_i = j-1} \{ \sum_{i=1}^f p(w_i, h_i) + \alpha(h_i) d(v, w_i) \}$ implicaria em se 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 , $f > 1$. Defina $T_v^i = T_v - \bigcup_{i < i \leq f} T_{w_i}$. Isto é, T_v^i é a subárvore parcial de raiz v , contendo as subárvores $T_{w_1}, \dots, T_{w_{i-1}}$ (figura 5.11). A idéia consiste em utilizar 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 , $f > 1$. 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 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 substituição 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

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

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. Conseqüentemente, não é mais necessário determinar as composições de $j - 1$, para computar $p(v, j)$. Na inicialização do processo, defina

$$p(v, j) = 0, \text{ se } j \leq 1$$

$p(v, j) = -\infty$, caso contrário.

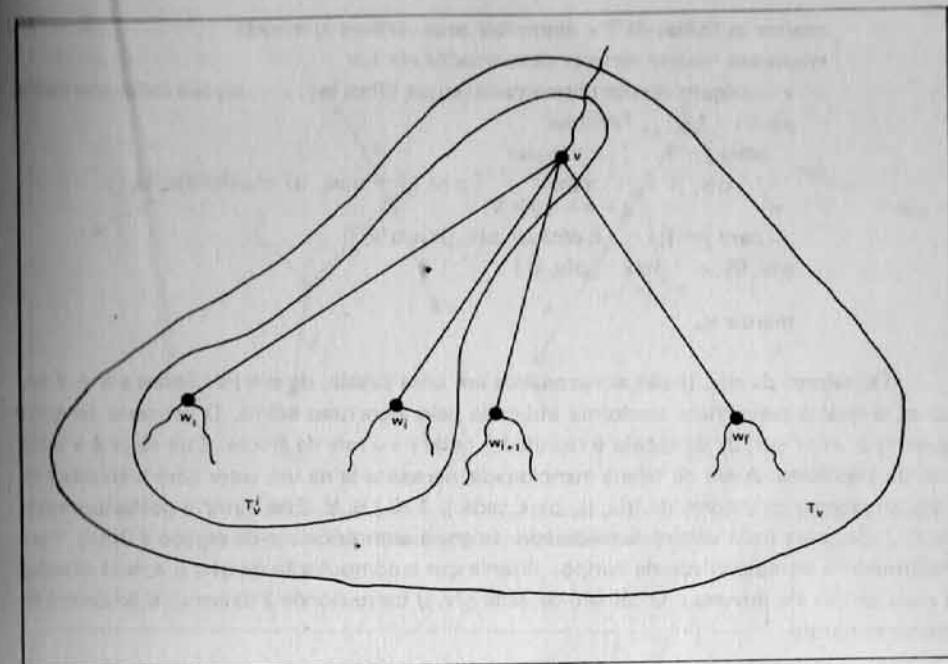


Figura 5.11. A subárvore parcial T_v^i

Essas condições iniciais já resolvem os subproblemas das folhas. Sejam 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 idéia consiste em supor que a subárvore T_v é construída passo a passo, através da incorporação a T_v das subárvores T_{w_i} para $1 \leq i \leq f$. No passo inicial, T_v consiste unicamente de v . Após cada incorporação de T_{w_i} a T_v , corrigem-se os valores de $p(v, j)$ obtidos. No caso geral, as subárvores $T_{w_1}, T_{w_2}, \dots, T_{w_{i-1}}$ 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 $p(v, j) := q(v, j)$

$p(v, 0) := \max_{1 \leq j \leq k} \{ p(v, j) \}$

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 acima. 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 com k elementos, para armazenar os valores de $q(v, j)$, 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 $p(v, j)$, para um certo v , podem ser calculados em $O(k^2)$ tempo. A complexidade de tempo é, portanto, $O(nk^2)$.

Como exemplo, seja particionar a árvore da figura 5.12(a), com $k = 3$. Escolheu-se (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 negativos). Sendo dado um inteiro k , deseja-se agora determinar uma partição P dos vértices da árvore, tal que a soma dos pesos

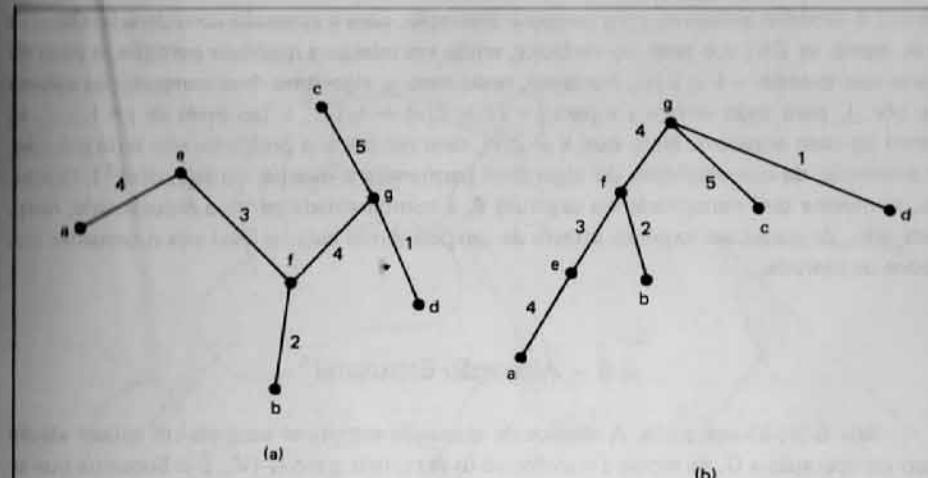


Figura 5.12. O exemplo correspondente à figura 5.13.

$v \setminus j$	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 $k = 3$

dos vértices em cada parte de P seja $\leq k$ 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 uni-

tários) é também aplicável, com 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 $\geq Z(v)$. 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.

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:

- 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 $G'(V - \{v\}, E')$, ou respectivamente $G'(V, E - \{e\})$. Como a transformação é simples, freqüentemente 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 .
- 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.
- 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.

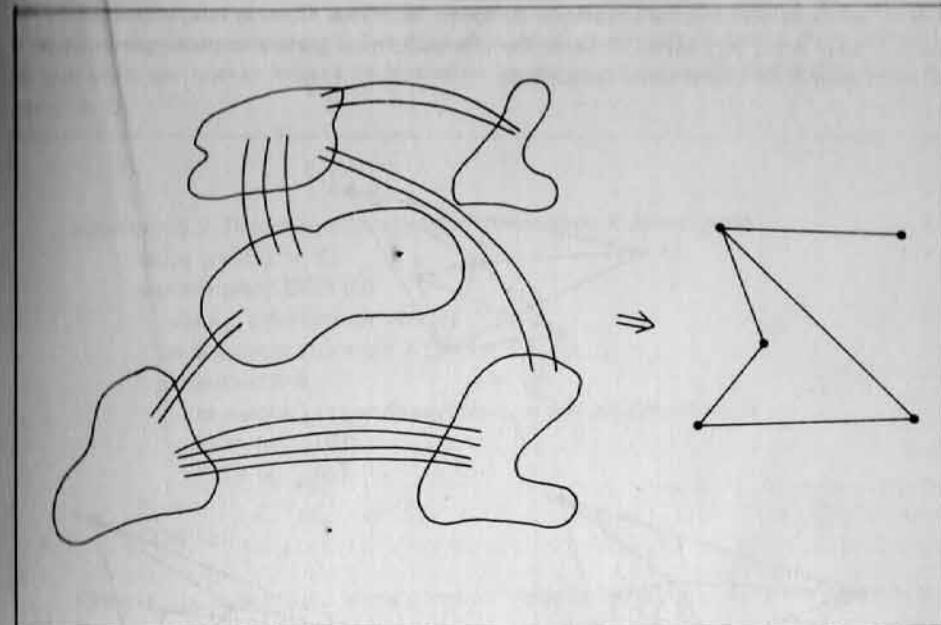


Figura 5.14. Operação de condensação

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 idéia 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 $(V, E \cup \{(v, w)\})$. $\beta_{v,w}(G)$ corresponde ao grafo 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 idéia do algoritmo para a obtenção do número cromático $X(G)$ de um grafo $G(V, E)$ é simples. Se G for completo, evidentemente $X(G) = |V|$. 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 $\beta_{v,w}(G)$, respectiva-

mente, aplica-se 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.

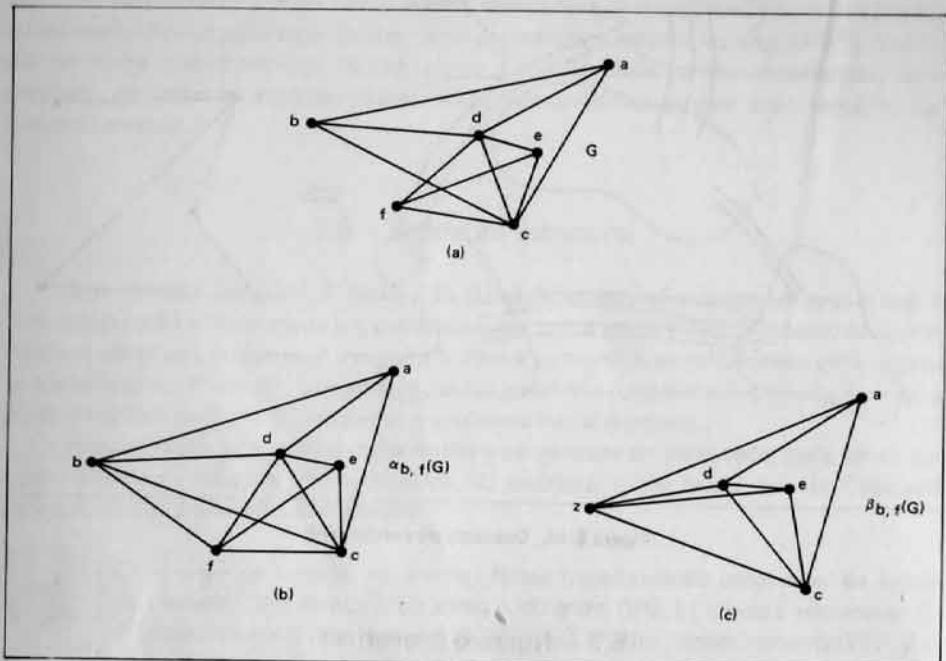


Figura 5.15. Os grafos α e β .

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 $X(G)$ satisfaz:

$$X(G) = \min \{ X(\alpha_{v,w}(G)), X(\beta_{v,w}(G)) \}$$

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, $X(G) = X(\alpha_{v,w}(G))$. 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, $X(G) = X(\beta_{v,w}(G))$. Logo, $X(G)$ deve ser o menor entre $X(\alpha_{v,w}(G))$ e $X(\beta_{v,w}(G))$. ▲

Observe ainda que pela aplicação recursiva do teorema acima 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 α e β , a partir de G .

algoritmo 5.3: Determinação do número cromático X 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 $X := \min \{ X, n_G \}$

caso contrário

encontrar um par de vértices v, w não adjacentes em G

$\text{COR}(\alpha_{v,w}(G))$

$\text{COR}(\beta_{v,w}(G))$

$X := n$

$\text{COR}(G)$

Observe que o algoritmo acima 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 preordem, onde a subárvore esquerda de cada vértice é totalmente construída, antes de se iniciar a da direita. A figura 5.16 mostra a árvore obtida para determinar o número cromático do grafo ilustrado na 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 $X(\alpha_{v,w}(G')) < X(\beta_{v,w}(G'))$, então a coloração ótima de G' será exatamente igual a de $\alpha_{v,w}(G')$. Se $X(\alpha_{v,w}(G')) > X(\beta_{v,w}(G'))$ entao 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 . No caso da igualdade dos números cromáticos de $\alpha_{v,w}(G')$ e $\beta_{v,w}(G')$, pode-se aplicar tanto um como o outro procedimento. A figura 5.16 ilustra também a coloração ótima obtida.

O número de vértices da árvore T pode ser exponencial em relação ao número de vértices do grafo G . Conseqüentemente, este algoritmo possui complexidade exponencial.

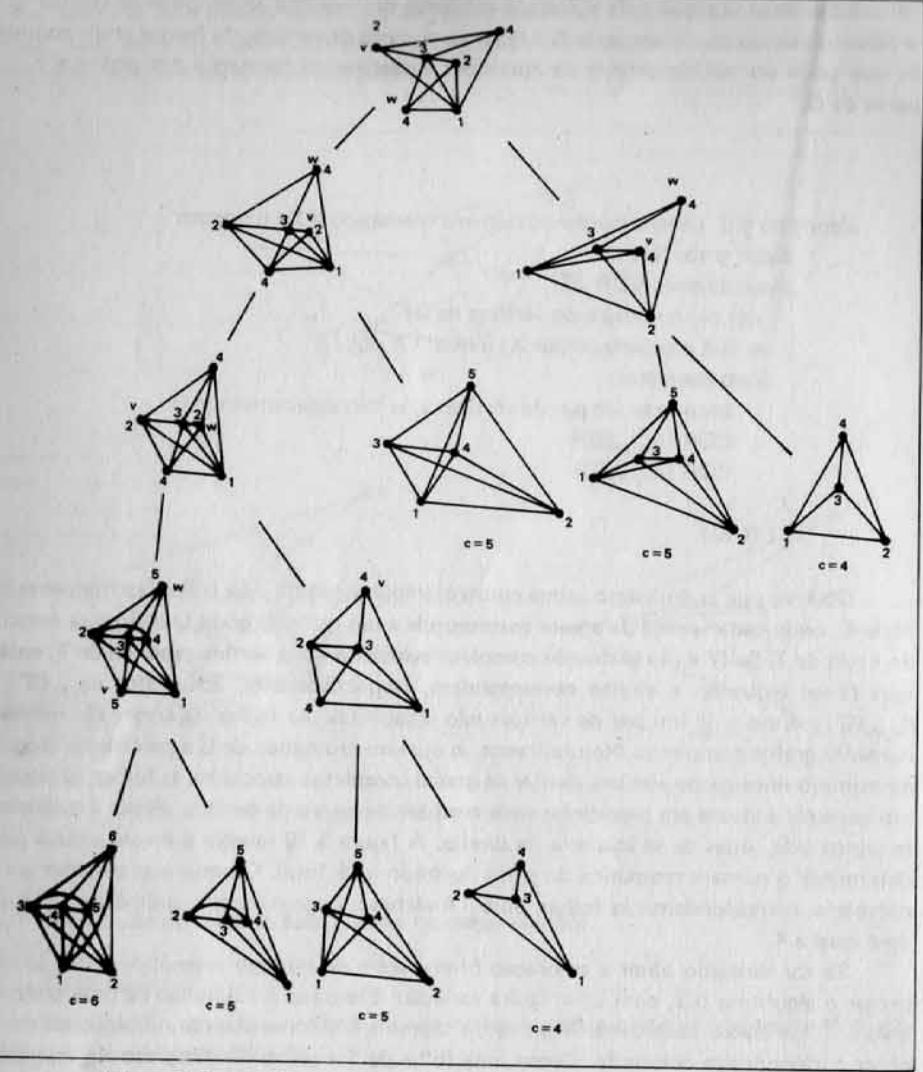


Figura 5.16. Algoritmo para determinação do número cromático de um grafo

5.8 – EXERCÍCIOS

- 5.1 Considere a seguinte variação do algoritmo guloso para determinação da árvore geradora máxima $T(V, E_T)$ 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 máximas são isomórficas entre si.
- 5.3 Caracterizar os grafos, com pesos nas arestas, para os quais todas as árvores geradoras máximas 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 partitionar 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 mínimo. A seguinte é uma tentativa de resolver este problema. "Obter uma árvore geradora máxima T , de G , utilizando o 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
- $$X(\alpha_v, w(G)) < X(\beta_v, w(G)),$$
- sendo v, w um par 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 $X(\alpha_{v'}, w(G')) \leq X(\beta_{v'}, w(G'))$. 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 , $X(G) \leq p \leq |V|$. Provar ou dar contra-exemplo.
- 5.13 Obter a expressão da complexidade do algoritmo 5.3 para determinação do número cromático de um grafo.

5.9 – NOTAS BIBLIOGRÁFICAS

O algoritmo 5.1, para determinar a árvore geradora máxima 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). Mais recentemente, algoritmos de complexidade $O(m\log \log n)$ foram apresentados em Yao (1975) e Chertton e Tarjan (1976). 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 Zikov (1949). Uma implementação deste algoritmo foi também realizada por Corneil e Graham (1973).

CAPÍTULO 6

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 variedade destes pode ser resolvida 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 seqüência de apresentação.

6.2 – O Problema do Fluxo Máximo

Uma *rede* é um multidígrafo $D(V, E)$ 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 acima 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 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 $f(D)$.

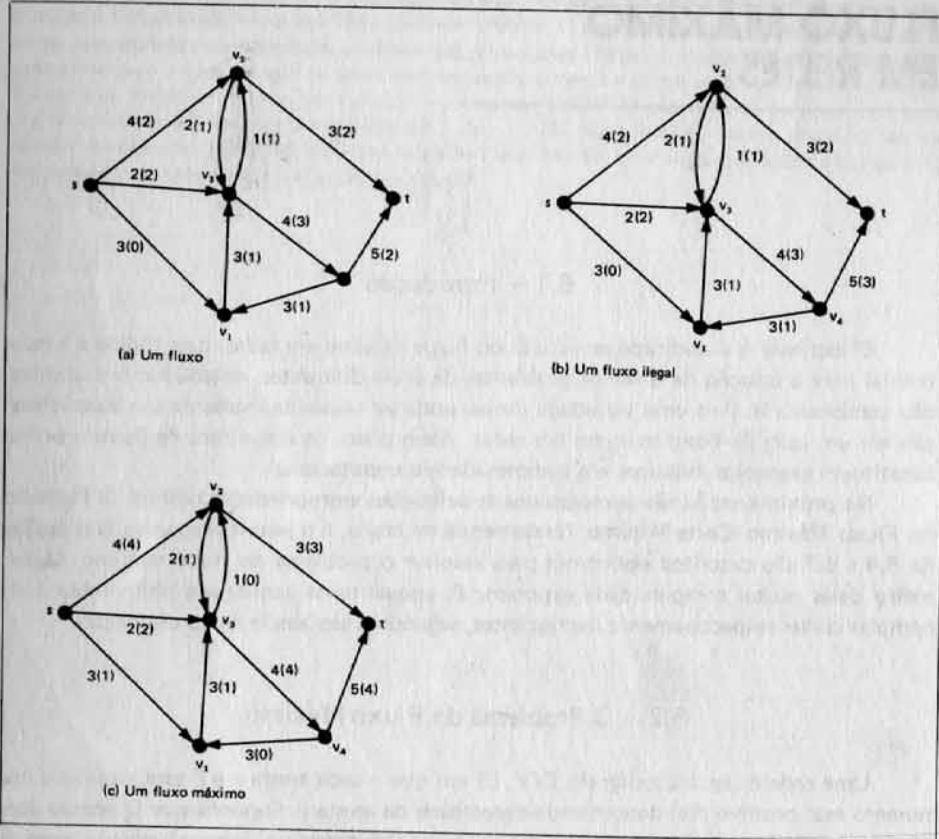


Figura 6.1. Fluxo em redes

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) acima (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 acima a atribuição f será chamada *fluxo ilegal*.

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) acima. 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.

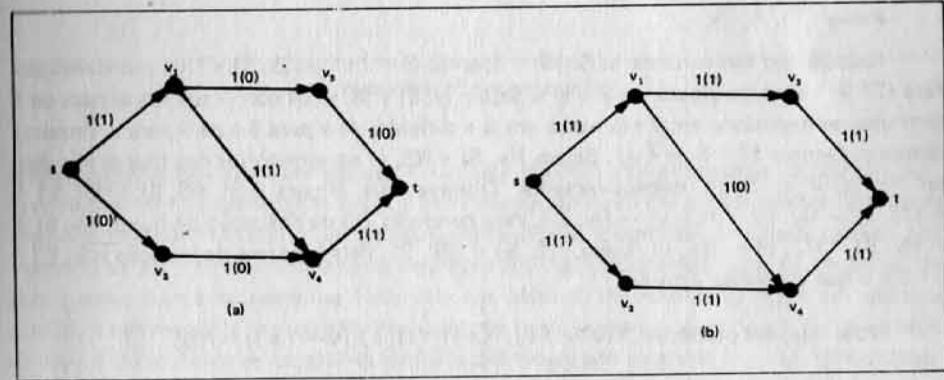


Figura 6.2. Fluxos maximal e máximo

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

$$(S, \bar{S})^+ = \{(v, w) \in E \mid v \in S \text{ e } w \in \bar{S}\}$$

$$(S, \bar{S})^- = \{(v, w) \in E \mid v \in \bar{S} \text{ e } w \in S\}$$

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 é definido como a diferença

$$f(S, \bar{S}) = \sum_{e \in (S, \bar{S})^+} f(e) - \sum_{e \in (S, \bar{S})^-}$$

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 $f(S, \bar{S}) = f(D)$.

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) $f(S, \bar{S}) = f(S', \bar{S}') + f(v, S) + f(S, v) - f(v, \bar{S}) - f(\bar{S}, v)$. 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. Esta caracterização será obtida com 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

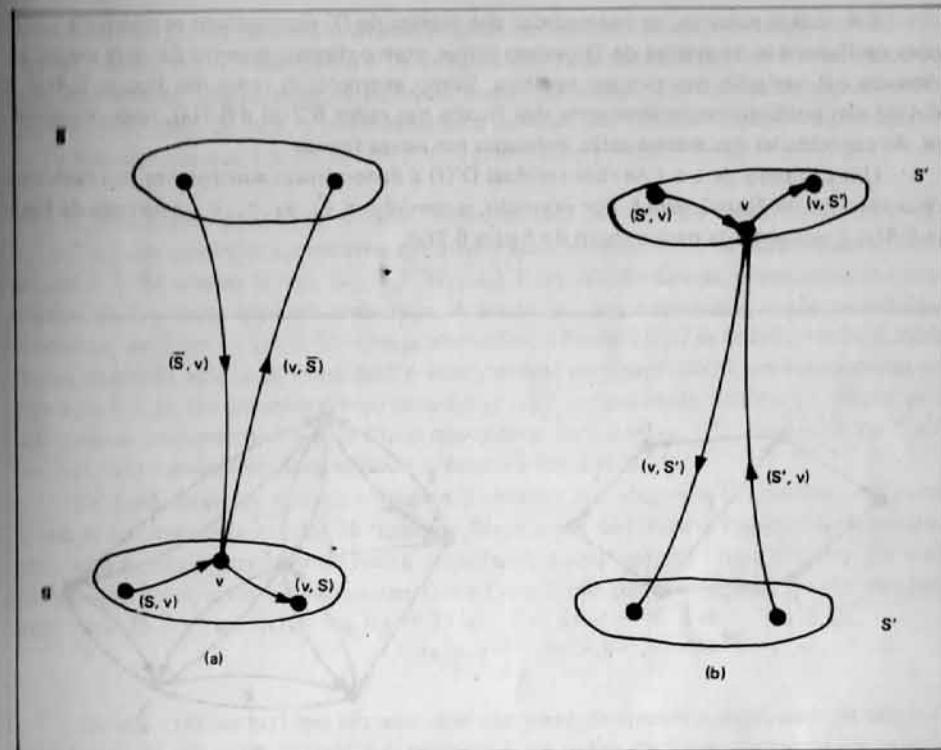


Figura 6.3. Prova do lema 6.1

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 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. 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).

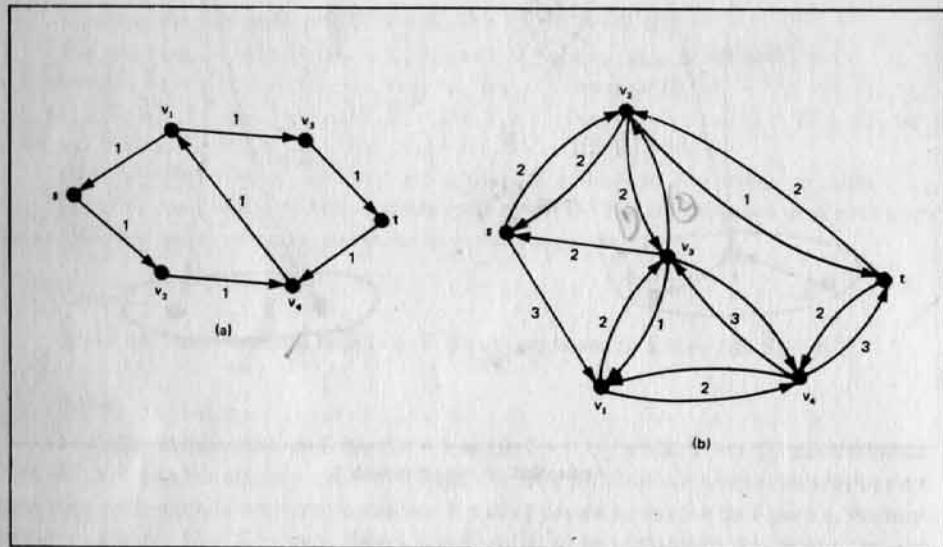


Figura 6.4. Redes residuais

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, de-

crementar $f(v_{j+1}, v_j)$ 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 acima, $f(D) \leq c_{\min}$, 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 $t \notin S$. 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 em 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

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

$F' := \min \{c'(v_j, v_{j+1}) \mid 1 \leq j < k\}$

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 acima não existir, o fluxo é 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 para $j = 1, \dots, k-1$ efetuar. 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.

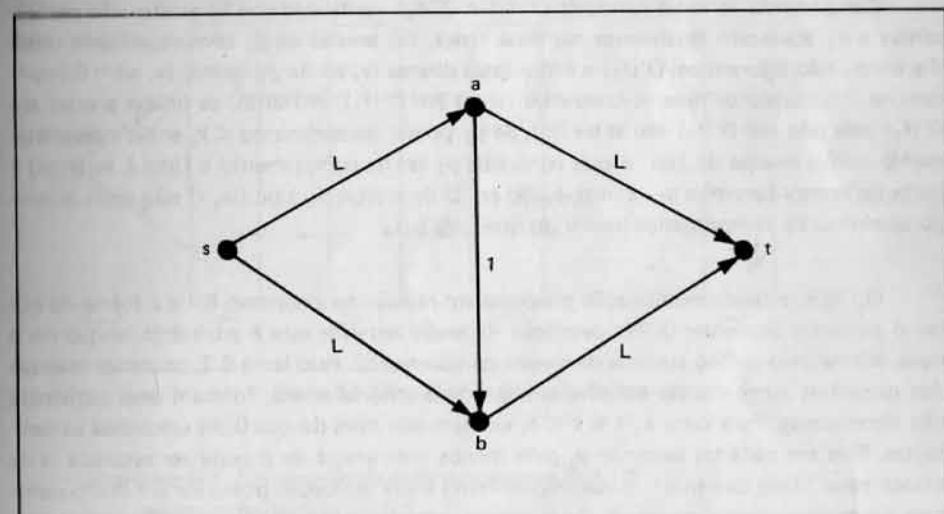


Figura 6.5. Um caso ruim para o algoritmo 6.1

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, o qual 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 na 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 acima formam uma seqüência não decrescente. Para cada k , $1 \leq k \leq n$, 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 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 acima 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 $O(n^2 m)$

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 da 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 $D'(f)$.

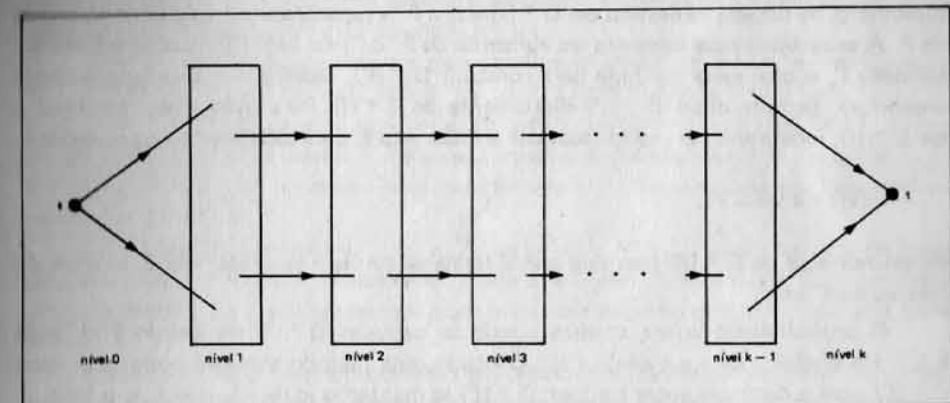


Figura 6.6. Esquema de uma rede de camadas

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 nível $(v) \geq$ nível (w) em T , então eliminar (v, w)

seja v_1, \dots, v_n a seqüência 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 ▲

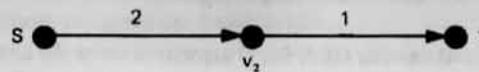


Figura 6.7. Rede de camadas para o fluxo da figura 6.1(a)

O algoritmo acima possui complexidade $O(m)$. A sua correção decorre do fato de que numa busca em largura não existe aresta (v, w) tal que 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).

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 dígrafo 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 dígrafo 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 das arestas. Isto é, para cada aresta e de p , efetuar

$$c'(e) := c(e) - F',$$

eliminando-se e de $D^*(f)$ caso esta aresta tenha se tornado saturada, isto é, se $c'(e)$ decresceu para zero.

O procedimento acima 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 acima 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 acima, 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.

algoritmo 6.3: Fluxo máximo em uma rede

```

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
    obter um fluxo maximal  $f^*$  de valor  $F^*$  em  $D^*$ 
    para  $e \in E$  efetuar
        se  $e$  está em  $D^*$  então  $f(e) := f(e) + f^*(e)$ 
         $F = F + F^*$ 
    construir  $D^*(f)$  pelo algoritmo 6.2

```

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 mais acima 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 acima encontra um fluxo maximal em D^* após $O(nm)$ passos. Consequentemente a complexidade do algoritmo para determinar um fluxo máximo em D é $O(n^2 m)$.

Como exemplo, seja determinar o fluxo máximo na rede D da figura 6.8(a). Na primeira iteração, constrói-se a rede residual D' (figura 6.8(b)) e a rede de camadas correspondente D^* (6.8(c)). Encontra-se um fluxo maximal f^* em D^* (6.8(c)). O fluxo em D é então aumentado do valor $f^*(D^*)$, como indica a figura 6.8(d). O processo se repete até que D^* não mais exista. (6.8(m)). O fluxo na rede D corrente (6.8(j)) é máximo.

6.7 — Um Algoritmo $O(n^3)$

Na presente seção apresenta-se um algoritmo para determinar um fluxo maximal em uma rede de camadas, o qual possui complexidade $O(n^2)$. Isto permite obter o fluxo máximo em uma rede em $O(n^3)$ passos, conforme já mencionado.

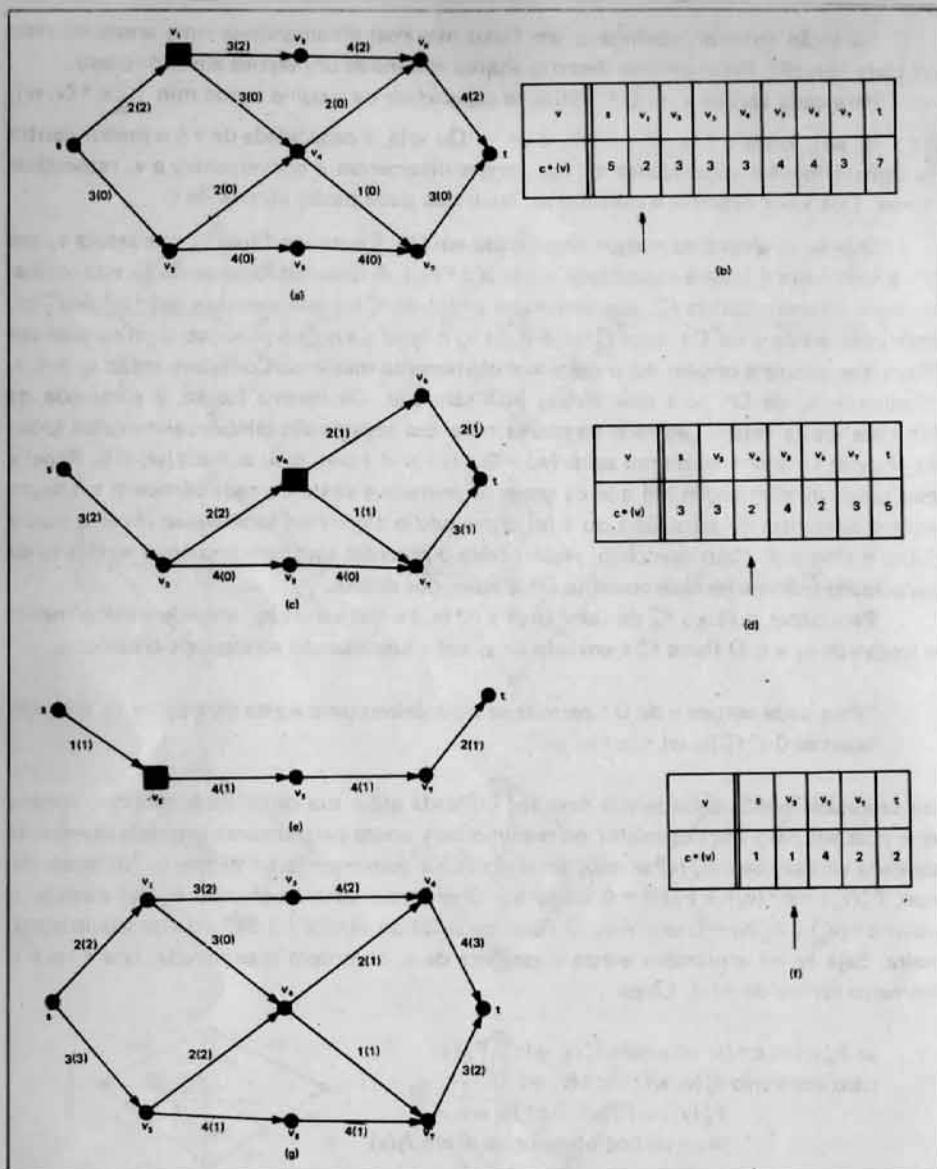


Figura 6.9. Um exemplo do algoritmo de fluxo maximal

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_v^* 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 $O(n^3)$ 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, seja obter um fluxo maximal na rede D^* da figura 6.9(a). Os valores das capacidades dos vértices estão indicados na figura 6.9(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 6.9(c), de onde se inicia uma nova iteração e assim por diante. O fluxo maximal obtido é o da figura 6.9(g).

6.8 – EXERCÍCIOS

- 8.1 Provar ou dar contra-exemplo.
Seja D uma rede. Em todo fluxo máximo em D o fluxo em cada aresta contrária é nulo.
- 8.2 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 $f(e) > 0$.
- 8.3 Provar que o valor do fluxo em qualquer corte de uma rede é não negativo.
- 8.4 Provar ou dar contra-exemplo.
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 $f(S, \bar{S}) = 0$.
- 8.5 Mostrar que se as capacidades das arestas de uma rede puderem assumir valores iracionais, 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 6.10.

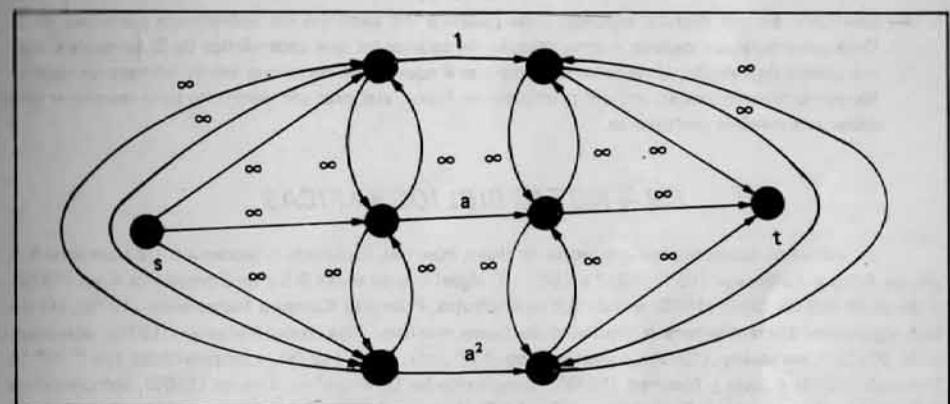


Figura 6.10. Sugestão do exercício 6.5

- 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

- (i) a conectividade em arestas de G .
- (ii) a conectividade em vértices de G .

Determinar a complexidade dos algoritmos.

- 6.12 Seja G um grafo bipartite 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 dígrafo. Uma *cobertura de 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 de ciclos. Em que condições existe a solução?
- 6.15 Seja $D(V, E)$ um dígrafo 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 dígrafo acíclico. Uma *cadeia* é um caminho no fechamento 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.

6.9 – 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, Pramod 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 $O(nm\log n)$. 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. 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).

CAPÍTULO 7

PROBLEMAS NP-COMPLETO

7.1 – 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 P. O interesse é conhecer se nessa coleção existe algum que seja eficiente, isto é, de complexidade polinomial. Se existir tal algoritmo, o problema P será denominado *tratável*, e *intratável* caso contrário. A idéia 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 7.2. A classe P, compreendendo os problemas tratáveis, é examinada na 7.3. Uma pequena lista de problemas aparentemente intratáveis é dada a seguir. A classe NP é o assun-

to da seção 7.5, enquanto que a questão $P = NP$ é apresentada em seguida. O complemento de um problema de decisão é o assunto da 7.1. Em seqüência, é introduzida a idéia de transformação polinomial entre problemas, a qual é utilizada para definir a classe NP-completo. Uma pequena lista desses é apresentada na seção 7.9. As restrições e extensões de problemas são descritas na 7.10, enquanto que o conceito de algoritmo pseudopolinomial encerra o capítulo.

7.2 – 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 retirados 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 acima 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 acima de encontrar no grafo G uma clique de tamanho $k > 0$. Uma possível instância do problema é o grafo da figura 7.1, 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.

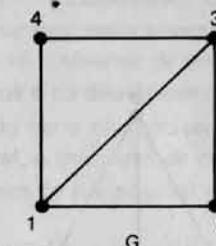


Figura 7.1. O grafo G e o inteiro 3 formam uma instância do problema de clique

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 7.2.

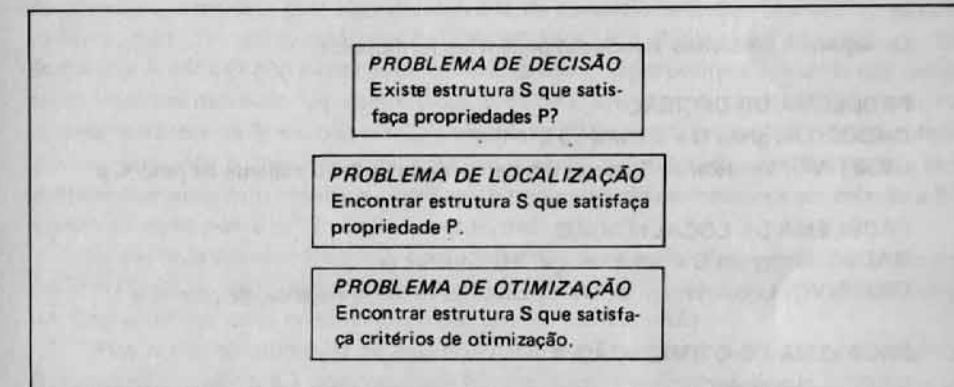


Figura 7.2. Problemas de decisão, localização e otimização associados

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 7.3, 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.

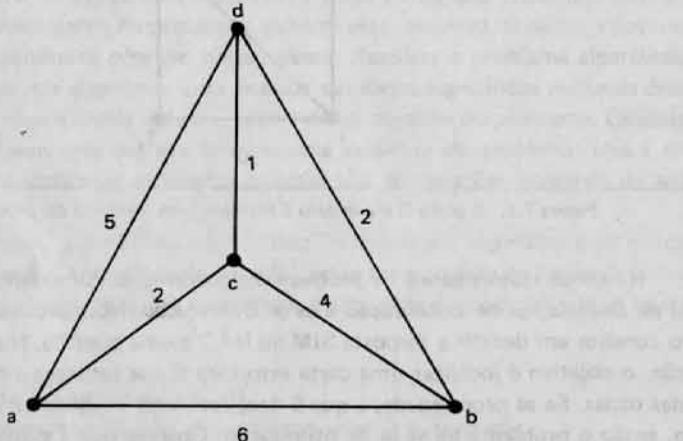


Figura 7.3. Problema do caixeiro viajante

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 Optimização* respeitivo seja resolvido e denote 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 *Optimização*. Aliás, é natural que assim o seja. Contudo é bastante menos intuitivo que, também em diversos casos, os problemas de otimização e localização apresentam 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 $\P(D, Q)$ será utilizada para representar um problema de decisão \P . D representa o conjunto de dados e Q a questão (decisão) correspondente. Quando conveniente, utiliza-se $\P(I)$ para denotar o problema $\P(D, Q)$ aplicado à instância $I \in D$.

7.3 – A Classe P

Conforme mencionado na seção 7.1, 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 acima 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 \P . 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 \P 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 acima, 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 acima de classificação de algoritmos quanto a sua eficiência foi também motivada por outros fatores. Por exemplo, as expressões de complexidade dos algoritmos polinomiais são freqüentemente 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 idéia de *passo* de um algoritmo é fundamental para a conceituação de sua complexidade. E esta idéia 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 acima, define-se a classe P de problemas de decisão como sendo aquela que comprehende precisamente aqueles 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 Π forem todos exponenciais, não necessariamente $\Pi \in P$. Se de fato $\Pi \notin P$ então deve existir alguma prova de que todo possível algoritmo para resolver Π 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.

7.4 – 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 abaixo, essa aparente intratabilidade.

PROBLEMA 1: SATISFABILIDADE

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 verdadeiro se e somente se \bar{x}_i for falso. De um modo geral, os 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 7.4 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 acima 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 acima é pois satisfatível. A expressão $(x_1) \wedge (\bar{x}_1)$ é obviamente não satisfatível, pois qualquer que seja o valor de x_1 , $(x_1) \wedge (\bar{x}_1)$ 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 7.4. Conjunção e disjunção

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 $(v, w) \in E$. Por exemplo, no grafo da figura 7.5, $\{a, c, b, g, h\}$ é um tal conjunto, de cardinalidade 5. De fato, a 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 vértice 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 7.5, $\{d, b, e\}$ 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.

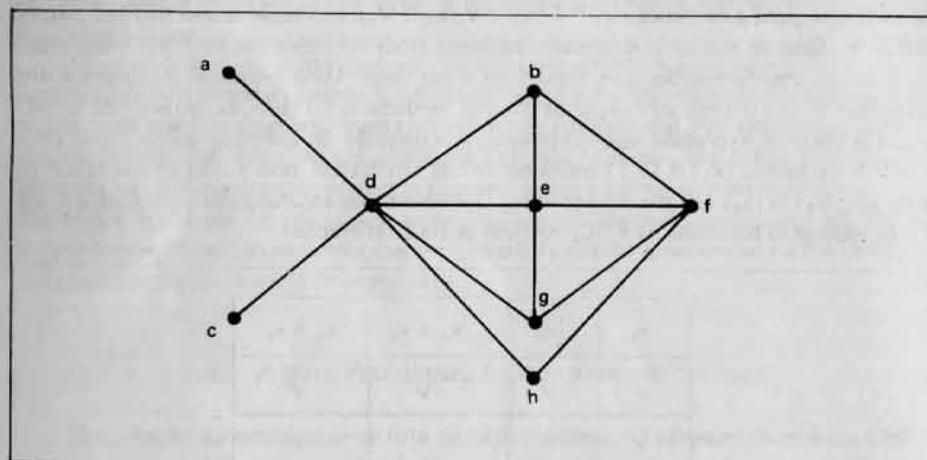


Figura 7.5. Exemplo para os problemas 2, 3 e 4

PROBLEMA 4: COBERTURA DE VÉRTICES

DADOS: Um grafo G e um inteiro $k > 0$

DECISÃO: G possui uma cobertura de vértices de tamanho $\leq k$?

Para um grafo $G(V, E)$, um subconjunto $V' \subseteq V$ é chamado *cobertura de 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 de vértices de tamanho 3, do grafo da figura 7.5. E, de fato, a cobertura de tamanho mínimo. O problema consiste em verificar se um grafo dado possui uma cobertura de vértices de tamanho no máximo igual a um valor dado.

PROBLEMA 5: CICLO HAMILTONIANO DIRECIONADO

DADOS: Dígrafo D

DECISÃO: D possui um ciclo hamiltoniano?

Recorde-se que um ciclo de um dado dígrafo D é dito *hamiltoniano* quando ele contém exatamente uma vez cada vértice de D . O ciclo a, b, d, f, g, e, c, a do dígrafo da figura 7.6(a) é obviamente hamiltoniano, enquanto que o dígrafo da figura 7.6(b) não possui tal ciclo. O presente problema consiste em reconhecer dígrafos 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: Dígrafo D e inteiro $k > 0$

DECISÃO: D possui um conjunto de arestas de realimentação de tamanho $\leq k$?

Para um dígrafo $D(V, E)$, 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 7.6(b), pode-se verificar que $E' = \{(a, d), (f, h), (b, e)\}$ é um conjunto de arestas de realimentação, pois todo ciclo do dígrafo exemplo contém pelo menos uma dessas três arestas. O presente problema consiste em verificar se um dado dígrafo possui um conjunto de arestas de realimentação de tamanho no máximo igual a um valor dado.

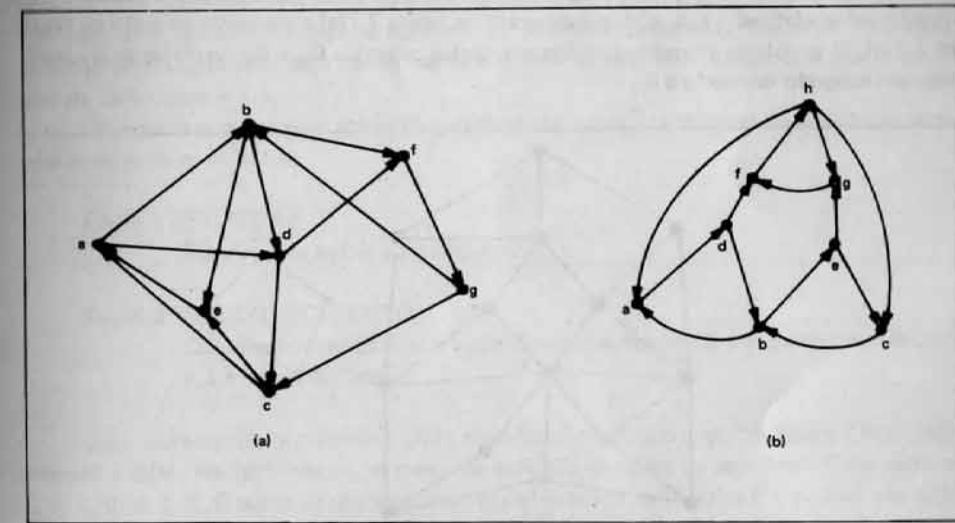


Figura 7.6. Exemplos para os problemas 5, 7, 8

PROBLEMA 8: CONJUNTO DE VÉRTICES DE REALIMENTAÇÃO

DADOS: Dígrafo 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 dígrafo $D(V, E)$ 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 dígrafo da figura 7.6(b) $V' = \{b, h\}$ é um conjunto de vértices de realimentação, de tamanho 2.

PROBLEMA 9: COLORAÇÃO

DADOS: Um grafo G e um inteiro $k > 0$

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 ?

Considere 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 7.7(a) é isomorfo ao grafo da figura 7.7(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 .

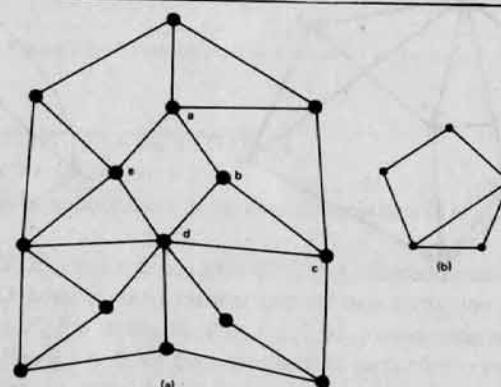


Figura 7.7. Exemplo para o problema 10

7.5 – A Classe NP

Considere um problema de decisão \P . Se \P for solúvel através da aplicação de algum processo, então necessariamente existe uma *justificativa* para a solução de \P . 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 correção do algoritmo.

Considere agora 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 ao problema *Ciclo Hamiltoniano* para o grafo da figura 7.8(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 o grafo da figura 7.8(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 de 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 acima de justificar respostas a problemas de decisão se compõe de duas fases distintas:

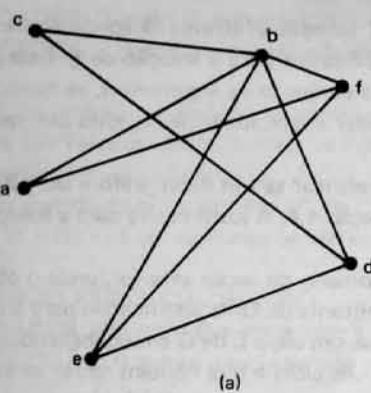
FASE 1: EXIBIÇÃO

Consiste em exibir a justificativa.

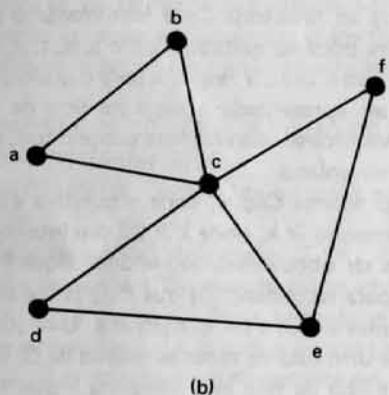
FASE 2: RECONHECIMENTO

Consiste em verificar que a justificativa apresentada (no passo de exibição) é, de fato, satisfatória.

Seja, novamente, o problema *Ciclo Hamiltoniano* para o grafo da figura 7.8(a), cuja solução é SIM. Na justificativa, o passo de exibição consiste da seqüência 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



Justificativa SIM: a, b, c, d, e, f, a
(passo de exibição)



Justificativa NÃO: a, b, c, a
e, c, d, e
e, c, f, e
e, d, c, f, e

Figura 7.8. Justificativa para resposta ao problema Ciclo Hamiltoniano

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 7.8(b), cuja solução é NÃO. O passo de exibição da justificativa é o conjunto das 4 seqüências: {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 seqüência de vértices é um ciclo não hamiltoniano e (ii) todo ciclo simples de G está presente na seqüência 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 idéia seria enumerar todos os ciclos do grafo G. Co-

mo 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, a 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 Π , tais que existe uma justificativa à resposta SIM para Π , cujo passo de reconhecimento pode ser realizado por um algoritmo polinomial no tamanho da entrada de Π .

Ressalte-se na definição da classe NP, que não se exige uma solução polinomial para Π . 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 7.8(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 Π 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 de J e da entrada de Π .

Se o algoritmo resultante do passo (2) for polinomial no tamanho da entrada de Π , então Π pertence a NP. O caso contrário, naturalmente, não implica necessariamente em não pertinência a NP.

Como ilustração do processo, mostra-se abaixo que os problemas Satisfabilidade, Conjunto Independente de Vértices e Cobertura de Vértices pertencem a NP. Aliás, todos os demais problemas apresentados na seção anterior também pertencem.

PROBLEMA: SATISFABILIDADE

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 se 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 SATISFABILIDADE. 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 DE VÉRTICES

DADOS: Um grafo $G(V, E)$ e um inteiro $k > 0$

DECISÃO: G possui uma cobertura de 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 DE VÉRTICES. Logo, pertence a NP.

Com essa mesma idéia, 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* abaixo.

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 acima pode ser assim descrita. No passo de exibição apresenta-se um conjunto S contendo todas as cliques máximas 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 só 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* é NP.

Finalmente, embora não considerados neste texto, menciona-se que existem problemas para os quais são conhecidas provas de não pertinência a NP.

7.6 – A Questão $P = NP$

Nessa seção discutem-se relações entre as classes P e NP. O lema seguinte é imediato.

Lema 7.1

$P \subseteq NP$.

Prova

Seja $\#$ um problema de decisão. Então existe um algoritmo α que apresenta a solução de $\#$, em tempo polinomial no tamanho de sua entrada. Em particular, α pode ser utilizado como algoritmo de reconhecimento para uma justificativa à resposta SIM de $\#$. Logo, $\# \in NP$.

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 7.8, serão desenvolvidos argumentos adicionais que reforçam a idéia 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 abaixo responde afirmativamente a esta pergunta.

Lema 7.2

Seja Π ∈ NP um problema de decisão. Então uma resposta SIM ou NÃO para Π pode ser obtida em tempo exponencial com o tamanho de sua entrada.

Prova

Se Π ∈ NP, então existe um algoritmo α que reconhece se é verdadeira uma justificativa J para a resposta SIM a Π , tal que α possui complexidade c_α , polinomial no tamanho da entrada de Π . Seja k o tamanho de J , o qual é necessariamente também polinomial na entrada de Π . 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 idéia 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 α . Então π possui resposta SIM se e somente se pelo menos uma das saídas de α for SIM. α é aplicado um total de $|A|^k$ vezes. Portanto, foi descrito um algoritmo para π de complexidade $O(|A|^k c_\alpha)$. Isto prova o lema. ▲

7.7 – 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 $\bar{\Pi}$, tais que existe uma justificativa à resposta NÃO, cujo passo de reconhecimento corresponde a um algoritmo polinomial no tamanho da entrada de $\bar{\Pi}$.

Por analogia, define-se o complemento $\bar{\Pi}$ de um problema de decisão Π como sendo o problema de decisão cujo objetivo é o complemento da decisão de Π . Ou seja, Π e $\bar{\Pi}$ possuem SIM e NÃO trocados. Isto é, a resposta ao problema Π 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, Π ∈ NP se e somente se $\bar{\Pi}$ ∈ Co-NP.

Pode-se formular para a classe Co-NP um lema semelhante ao 7.1. Ou seja, é imediato verificar que $P \subseteq \text{Co-NP}$. Assim sendo, se um problema de decisão Π admitir solução polinomial, então obviamente $\Pi \in P$ e $\Pi \in \text{NP} \cap \text{Co-NP}$. Por outro lado, existem problemas π e $\bar{\Pi}$ 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 pertinê-

cia 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 Π quanto do complemento $\bar{\Pi}$.

Como exemplo, considere o problema CLIQUE, anteriormente mencionado. Seu complemento é o problema CLIQUE seguiente:

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 CLIQUE ∈ 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 acima. 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. Conseqüentemente, não é sabido se CLIQUE ∈ NP.

Um exemplo para o qual é desconhecida a pertinência a NP tanto de Π , quanto de $\bar{\Pi}$ é a Clique Máxima, formulada na seção 7.5. Exemplos de problemas tais que ambos Π e $\bar{\Pi}$ pertencem a NP são inúmeros. Como foi mencionado, $P \subseteq \text{NP}$ e $P \subseteq \text{Co-NP}$. Logo, qualquer problema Π ∈ P satisfaz $\Pi \in \text{NP}$ e $\bar{\Pi} \in \text{NP}$. Há, também, outros para os quais se desconhece se $\Pi \in P$, mas se conhece que ambos Π , $\bar{\Pi}$ ∈ NP. Um exemplo desse último caso é o problema Números Compósitos, abaixo:

PROBLEMA: NÚMEROS COMPOSTOS

DADOS: Um número inteiro $k > 0$

DECISÃO: Existem inteiros $p, q > 1$ tais que $k = pq$?

É imediato verificar que NÚMEROS COMPOSTOS ∈ 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?

Embora não seja trivial, é conhecido um algoritmo que reconhece uma justificativa SIM para o problema acima, em tempo polinomial com a entrada. Isto permite concluir que o mesmo pertence à classe NP. Contudo, não se conhece, até o momento, algoritmo que possa resolver o problema em tempo polinomial. Em resumo, ambos NÚMEROS COMPOSTOS e NÚMEROS PRIMOS pertencem a NP. Contudo, não é sabido se pertencem ou não a P.

As seguintes questões não foram resolvidas até o momento:

- (i) $\text{NP} = \text{Co-NP}$?

(ii) $P = NP \cap Co\text{-}NP$?

Observe que as expressões (i) e (ii) acima não são independentes da questão $P = NP$? De fato, se $P = NP$, então necessariamente $NP = Co\text{-}NP$ e portanto ambas (i) e (ii) possuem resposta SIM. Contudo, outras combinações são também possíveis. Conjetura-se que de fato $NP \neq Co\text{-}NP$ e $P \neq NP \cap Co\text{-}NP$. A figura 7.9 ilustra as conjecturas.

7.8 – Transformações Polinomiais

Sejam $\mathbb{P}_1(D_1, Q_1)$ e $\mathbb{P}_2(D_2, Q_2)$ problemas de decisão. Suponha que seja conhecido um algoritmo A_2 para resolver \mathbb{P}_2 . Se for possível transformar o problema \mathbb{P}_1 em \mathbb{P}_2 e sendo conhecido um processo de transformar a solução de \mathbb{P}_2 numa solução de \mathbb{P}_1 , então o algoritmo A_2 pode ser utilizado para resolver o problema \mathbb{P}_1 . Observe que como o objetivo consiste em resolver \mathbb{P}_1 , através de A_2 , o que se supõe dado é uma instância de \mathbb{P}_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 \mathbb{P}_2 . O retorno ao problema \mathbb{P}_1 é realizado através da transformação da solução de \mathbb{P}_2 para a de \mathbb{P}_1 . Se a transformação de \mathbb{P}_1 em \mathbb{P}_2 , bem como a da solução de \mathbb{P}_2 na de \mathbb{P}_1 , puder ser realizada em tempo polinomial, então diz-se que existe uma *transformação polinomial* de \mathbb{P}_1 em \mathbb{P}_2 , e que \mathbb{P}_1 é *polinomialmente transformável* em \mathbb{P}_2 .

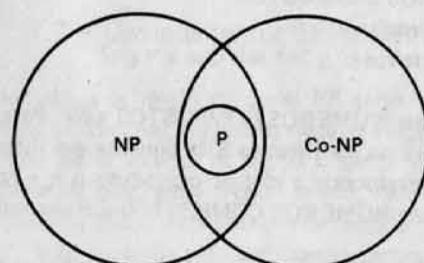


Figura 7.9. Conjecturas
 $P \neq NP$
 $NP \neq Co\text{-}NP$
 $P \neq NP \cap Co\text{-}NP$

Formalmente, uma *transformação polinomial* de um problema de decisão $\mathbb{P}_1(D_1, Q_1)$ no problema de decisão $\mathbb{P}_2(D_2, Q_2)$ 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 \mathbb{P}_1 , tem-se: $\mathbb{P}_1(I)$ possui resposta SIM se e somente se $\mathbb{P}_2(f(I))$ também o possui.

A figura 7.10 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 \mathbb{P}_2 , quando utilizado para resolver \mathbb{P}_1 . Assim sendo, se A_2 for polinomial, e existir uma transformação polinomial de \mathbb{P}_1 em \mathbb{P}_2 , então \mathbb{P}_1 também pode ser resolvido em tempo polinomial. Denota-se $\mathbb{P}_1 \leq_p \mathbb{P}_2$ para indicar que \mathbb{P}_1 pode ser transformado polinomialmente em \mathbb{P}_2 . Observe que a relação \leq_p é transitiva (isto é, $\mathbb{P}_1 \leq_p \mathbb{P}_2$ e $\mathbb{P}_2 \leq_p \mathbb{P}_3 \Rightarrow \mathbb{P}_1 \leq_p \mathbb{P}_3$).

Para ilustrar um processo simples de transformação polinomial, considere como \mathbb{P}_1 e \mathbb{P}_2 , respectivamente, os problemas CLIQUE e CONJUNTO INDEPENDENTE DE VÉRTICES, formulados na seção 7.4. A instância I de CLIQUE consiste de um grafo $G(V, E)$ e um inteiro $k > 0$. Para obter a instância $f(I)$ de CONJUNTO INDEPENDENTE, considera-se o grafo completo \bar{G} de G e o mesmo inteiro k . É então evidente que f é uma transformação polinomial porque

- (i) \bar{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 \bar{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 $CLIQUE \leq_p CONJUNTO\ INDEPENDENTE$.

Observe que quando $\mathbb{P}_1 \leq_p \mathbb{P}_2$ a transformação f deve ser aplicada sobre uma instância I genérica de \mathbb{P}_1 . A instância que se obtém de \mathbb{P}_2 contudo é particular, pois é fruto da transformação f utilizada. Assim sendo, de certa forma, \mathbb{P}_2 pode ser considerado como um problema de dificuldade maior ou igual a \mathbb{P}_1 . Pois que, mediante f , um algoritmo que resolve a instância particular de \mathbb{P}_2 , obtida através de f , resolve também a instância arbitrária de \mathbb{P}_1 dada.

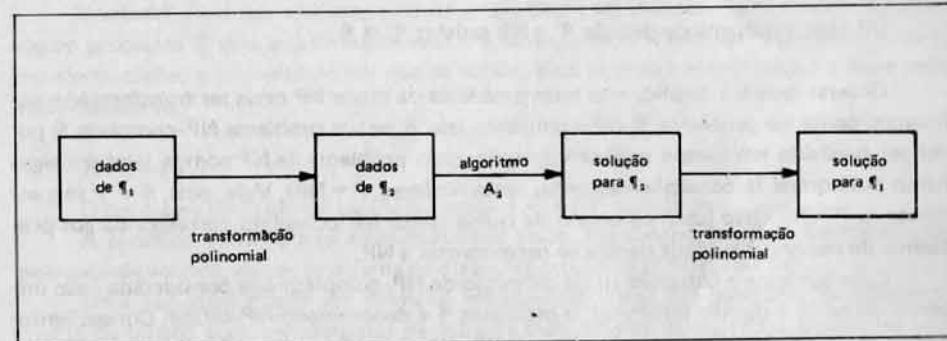


Figura 7.10. Transformação polinomial do problema \mathbb{P}_1 em \mathbb{P}_2

Considere agora dois problemas Π_1 e Π_2 , tais que $\Pi_1 \leq \Pi_2$ e $\Pi_2 \leq \Pi_1$. Então Π_1 e Π_2 são denominados *problemas equivalentes*. Segue-se que, segundo a observação acima, dois problemas equivalentes são de idêntica dificuldade (no que se refere à existência ou não de algoritmo polinomial para resolvê-los). Note que quando $\Pi_1 \leq \Pi_2$, a particularização de Π_2 construída pela transformação f é equivalente ao problema geral Π_1 . Isto é, representado 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 7.11.

Observe que é possível utilizar a relação \leq 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.

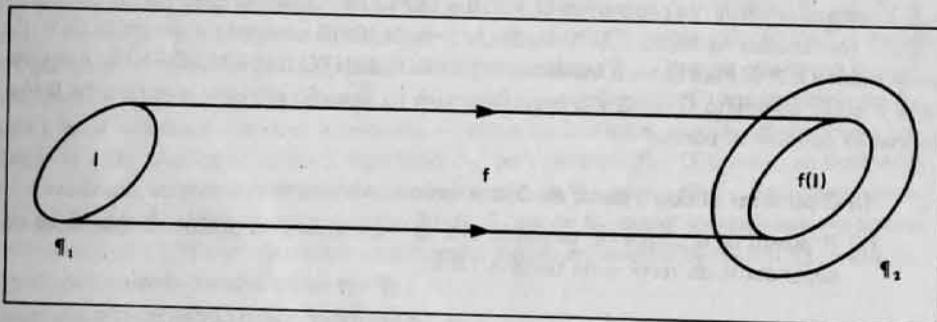


Figura 7.11. Equivalência entre $\Pi_1(D_1, Q_1)$ e $(f(D_1), Q_2)$

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-completo*, cuja definição se segue.

Um problema de decisão Π 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 $\Pi' \leq \Pi$.

Observe que (ii) implica que todo problema da classe NP pode ser transformado polynomialmente no problema Π NP-completo. Isto é, se um problema NP-completo Π puder ser resolvido em tempo polinomial então *todo* problema de NP admite também algoritmo polinomial (e consequentemente, nesta hipótese, $P = NP$). Vale, pois, $\Pi \in P$ se e somente se $P = NP$. 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 (i) da definição de NP-completo seja considerada (não importando se (ii) é ou não satisfeita), o problema Π é denominado *NP-difícil*. Consequentemente, a "dificuldade" de um problema NP-difícil é não menor do que a de um NP-completo.

7.9 – Alguns Problemas NP-Completo

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) acima requer que seja provado que *todas* problemas de NP podem ser polynomialmente transformados 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 7.3

Sejam Π_1, Π_2 problemas de decisão em NP. Se Π_1 é NP-completo e $\Pi_1 \leq \Pi_2$ então Π_2 é também NP-completo.

Prova

Como $\Pi_2 \in \text{NP}$, para mostrar que Π_2 é NP-completo, basta provar que $\Pi' \leq \Pi_2$, para todo $\Pi' \in \text{NP}$. Como Π_1 é NP-completo, então necessariamente $\Pi' \leq \Pi_1$. Como $\Pi_1 \leq \Pi_2$, por transitividade tem-se $\Pi' \leq \Pi_2$.

O lema acima, apesar da simplicidade, é muito poderoso. Como consequência, para provar que um certo problema Π é NP-completo, ao invés de utilizar a mencionada condição (ii) da definição (a qual exige uma prova de que todo problema $\Pi' \in \text{NP}$ é polynomialmente transformável em Π), basta demonstrar que *um* problema NP-completo é polynomialmente transformável em Π . Ou seja, é suficiente provar que

- (i) $\Pi \in \text{NP}$, e
- (ii) um problema NP-completo Π' é tal que $\Pi' \leq \Pi$.

Observe que se apenas (ii) for realizado, então Π é NP-difícil, mas não necessariamente NP-completo.

Contudo, para que este esquema de prova possa ser utilizado, é necessário escolher algum problema Π' que seja NP-completo. Portanto, para se identificar o *primeiro* problema desta classe, o processo acima não se aplica. Essa primeira identificação é dada pelo teorema abaixo.

Teorema 7.1

O problema SATISFABILIDADE (seção 7.4) é NP-completo.

A prova do teorema foge ao escopo desse texto. Ela consiste em uma transformação polynomial genérica, de um problema da classe NP em SATISFABILIDADE.

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 acima. Assim sendo, tem-se:

Lema 7.4

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 é NP. É necessário agora verificar que algum problema NP-completo pode ser polinomialmente transformável em CLIQUE. Seja uma instância genérica de SATISFABILIDADE. 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 em G , para cada ocorrência 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 $x_i \neq \bar{x}_j$, 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 7.12). Para completar a prova, basta observar que a construção de G pode ser facilmente realizada em tempo polinomial com o tamanho de E .

Lema 7.5

O problema CONJUNTO INDEPENDENTE é NP-completo.

Na seção 7.5 foi provado que CONJUNTO INDEPENDENTE é NP. Na seção 7.8 foi verificado que CLIQUE \Leftrightarrow CONJUNTO INDEPENDENTE. Sabe-se pelo lema 7.4 que CLIQUE é NP-completo. Logo, CONJUNTO INDEPENDENTE é NP-completo.

Lema 7.6

COBERTURA DE VÉRTICES é NP-completo.

Prova

É imediato verificar que COBERTURA DE VÉRTICES é NP. Considere uma instância genérica de CONJUNTO INDEPENDENTE, o qual é NP-completo (lema 7.5). Esta se compõe de um grafo G e de um inteiro $k > 0$. Seja transformar esta instância numa de COBERTURA DE VÉRTICES, cuja entrada consiste de 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 DE 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 de vértices de G , de tamanho $\leq p$. Reciprocamente, se C é uma cobertura de vértices tal que $|C| \leq p$, então obviamente $V - C$ é um conjunto independente de tamanho $\geq k$. Logo, CONJUNTO INDEPENDENTE \Leftrightarrow COBERTURA DE VÉRTICES.

Seguindo uma seqüência similar de passos, pode-se mostrar que todos os problemas enunciados na seção 7.4 são NP-completo. 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 $P \neq NP$.

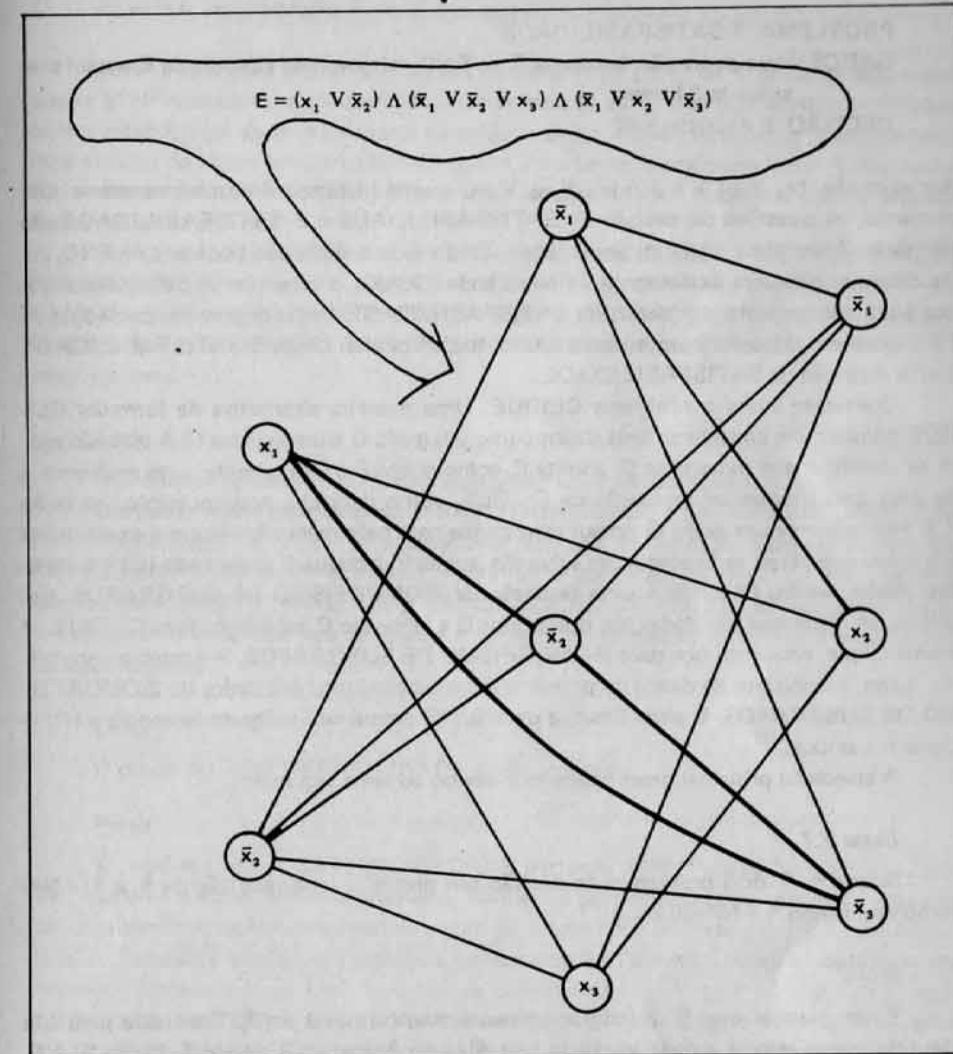


Figura 7.12. Transformação da prova do lema 7.4

7.10 – Restrições e Extensões de Problemas

Sejam $\pi(D, Q)$ e $\pi'(D', Q')$ problemas de decisão. Diz-se que π' é uma *restrição* de π quando $D' \subseteq D$ e $Q' = Q$. Analogamente, o problema Π' é chamado de *extensão* de Π . Isto é, a questão a ser respondida para ambos os problemas é a mesma. Mas os dados de Π' são retirados de um conjunto D' igual ou mais restrito do que o conjunto D , de onde os dados de Π são retirados.

Como exemplo, considere o problema abaixo.

PROBLEMA: 3-SATISFABILIDADE

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 acima. Obviamente, as questões de decisão de SATISFABILIDADE e 3-SATISFABILIDADE 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 SATISFABILIDADE (cujas expressões booleanas na FNC possuem cláusulas com número arbitrário de literais). Logo, 3-SATISFABILIDADE é uma restrição de SATISFABILIDADE.

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 7.2. 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 7.7

Sejam Π, Π' dois problemas de decisão tais que Π' é uma restrição de Π , e Π' é NP-completo. Então Π é NP-difícil.

Prova

Basta mostrar que Π' é polinomialmente transformável em Π . Considere para f , a identidade que associa a cada instância I de Π' , uma imagem $f(I) = I$ em Π . Então $\Pi' \leq \Pi$, o que prova o lema. ▲

Naturalmente, se no lema acima Π pertencer a NP então Π' é NP-completo. Observe também que quando o problema de decisão Π' for uma restrição de Π então um algoritmo polinomial que resolva Π , resolverá também Π' . Por outro lado, o fato de Π' ser NP-completo não implica, necessariamente, em que Π também o seja.

O lema 7.7 sugere um método alternativo para se provar que um certo problema Π é NP-completo. Ou seja, Π será NP-completo quando:

- (i) Π pertencer à classe NP, e
- (ii) existir uma restrição Π' de Π , que seja NP-completo.

Observe que para provar (ii), a idéia consiste em a partir de Π procurar algum problema Π' NP-completo, e tal que Π' seja uma restrição de Π . 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 Π . Muitas vezes, a alternativa acima 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 lema 7.4, 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.

Lema 7.8

O problema ISOMORFISMO DE SUBGRAFOS é NP-completo.

Considere agora os problemas 3-SATISFABILIDADE e SATISFABILIDADE. Conforme mencionado, o primeiro desses é uma restrição do segundo. Sabe-se também que SATISFABILIDADE é NP-completo. Portanto, a aplicação do lema 7.7 nada permite afirmar com relação a conhecer se 3-SATISFABILIDADE é ou não NP-completo. Assim sendo, para verificar que este problema é de fato NP-completo, utiliza-se a metodologia da seção anterior.

Lema 7.9

O problema 3-SATISFABILIDADE é NP-completo.

Prova

É imediato que 3-SATISFABILIDADE pertence a NP. Porque SATISFABILIDADE, que é uma extensão desse problema, também o pertence. O passo seguinte é estabelecer uma transformação polinomial de algum problema NP-completo em 3-SATISFABILIDADE. Considere então uma instância genérica de SATISFABILIDADE, 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-SATISFABILIDADE, que é uma restrição de SATISFABILIDADE, tal que cada cláusula possui exatamente 2 literais. Novamente, a aplicação do lema 7.7 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-SATISFABILIDADE pertence a P. Considere agora o problema k-SATISFABILIDADE, que é a restrição de SATISFABILIDADE, em que cada cláusula possui exatamente k literais. Como 3-SATISFABILIDADE é NP-completo, a aplicação do lema 7.7 permite concluir que k-SATISFABILIDADE também é NP-completo.

7.11 – Algoritmos Pseudopolinomiais

Na seção 5.5, foi apresentado um algoritmo de programação dinâmica para o problema de 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 de T em subárvores disjuntas, $T_1(V_1, E_1), \dots, T_t(V_t, E_t)$, tal que:

- (i) $\sum_{v \in V_i} z(v) \leq k$, para cada $i = 1, \dots, t$, e
- (ii) $\sum_{e \in E_i} d(e) \leq m$, 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 subárvores disjuntas T_1, \dots, T_t , de tal modo que (i) a soma dos pesos dos vértices em cada subárvore T_i não exceda um valor k dado e (ii) a soma dos pesos das arestas não pertencentes a qualquer das subárvores não excede um valor m dado. Em relação a este problema pode-se provar o seguinte:

Lema 7.10

O problema PARTICIONAMENTO DE ÁRVORES é NP-completo. A prova segue a estratégia geral apresentada na seção 7.9, mas será omitida neste texto.

O problema de otimização associado ao PARTICIONAMENTO DE ÁRVORES, acima 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 árvo-

re em subárvores, 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 5.6) 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, recordase 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 seqüência de $1 + \lfloor \log_2 k \rfloor$ dígitos. Conseqüentemente, $O(nk^2)$ 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-completo 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 Π um problema de decisão. Um algoritmo A que resolva Π é dito *pseudopolinomial* quando a complexidade de A for polinomial no tamanho da entrada de Π , 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 resolvê-lo é denominado *NP-completo fraco*. Assim sendo, PARTICIONAMENTO DE ÁRVORES é um exemplo de NP-completo fraco. Por outro lado, existem problemas NP-completo cuja possível existência de algoritmos pseudopolinomiais implicaria na igualdade $P = NP$. Esses últimos são chamados *NP-completo 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 de 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-completo, com tal característica, são NP-completo 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-completo fraco (como PARTICIONAMENTO DE ÁRVORES), bem como os NP-completo forte (como CAIXEIRO VIAJANTE).

Observe que um algoritmo pseudopolinomial para um problema \mathbb{P} 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 de 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.

7.12 – EXERCÍCIOS

7.1 Sejam A, B dois problemas tais que $A \in \text{NP}$ e $B \notin \text{NP}$. Então B é polynomialmente transformável em A se e somente se $P = \text{NP}$.

Certo ou errado?

7.2 Se $P = \text{NP}$ todo problema NP-difícil é polinomial.

Certo ou errado?

7.3 Provar que o problema CICLO HAMILTONIANO é NP-completo.

7.4 Usando o resultado do exercício anterior, provar que o problema de decisão CAIXEIRO VIAJANTE é NP-completo.

7.5 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 $\leq k$?

Provar que esse problema é NP-completo.

7.6 Provar que o problema CAMINHO MÍNIMO CONDICIONADO do exercício 7.5

(i) permanece NP-completo quando o grafo G é substituído por um dígrafo D , e

(ii) torna-se polinomial quando D é um dígrafo acíclico (nesse caso, apresentar um algoritmo polinomial para resolver o problema e determinar a sua complexidade).

7.7 O problema CAMINHO MÍNIMO CONDICIONADO do exercício 7.5 permanece NP-completo quando os pesos das arestas do grafo são todos unitários. Provar ou apresentar algoritmo polinomial.

- 7.8 Provar que o problema de determinar a árvore geradora da altura máxima de um grafo G é NP-difícil (e a de altura mínima?).
- 7.9 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.
- 7.10 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?
- 7.11 Seja A um problema polynomialmente transformável em B . Se B admite algoritmo pseudopolynomial para resolvê-lo, A também o admite.
Certo ou errado?
- 7.12 Se o complemento $\bar{\mathbb{P}}$ de todo problema $\mathbb{P} \in \text{NP}$ for tal que $\bar{\mathbb{P}}$ também esteja em NP, então:
(i) $\text{NP} = \text{Co-NP}$
(ii) $\text{P} = \text{NP}$
Certo ou errado?
- 7.13 Um problema de decisão \mathbb{P} é Co-NP-completo, quando:
(i) $\mathbb{P} \in \text{Co-NP}$, e
(ii) todo problema de decisão $\mathbb{P}' \in \text{Co-NP}$ satisfaz $\mathbb{P}' \leq \mathbb{P}$.
Então as classes NP-completo e Co-NP-completo são disjuntas se e somente se $P \neq \text{NP}$.
Certo ou errado?

7.13 – NOTAS BIBLIOGRÁFICAS

Os trabalhos pioneiros na teoria do NP-completo são Cook (1971) e Karp (1972). Foi Cook (1971) quem apontou o primeiro problema NP-completo (teorema 7.1). Um livro dedicado ao assunto é Garey e Johnson (1979), o qual contém também uma lista com centenas de problemas NP-completo. 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 7.4 foram todos retirados dos citados trabalhos pioneiros a saber: SATISFABILIDADE, 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). A justificativa SIM em tempo polinomial para o problema dos números primos é de Pratt (1975). Os 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 atualizadas sobre o que vem ocorrendo na área do NP-completo podem ser obtidas através da leitura da coluna de Johnson (1981).

- Busacker, R. G. and Saaty, T. L. (1965), *Finite graphs and networks: an introduction with applications*, McGraw-Hill, New York.
- Capobianco, M. and Molluzzo, J.C. (1978), *Examples and counter examples in graph theory*, North-Holland, New York.
- Carré, B. (1979), *Graphs and networks*, Clarendon Press, Oxford.
- Carvalho, R.L. (1981), *Máquinas, programas e algoritmos*, 2^a Escola de Computação, Inst. de Mat., Estat. 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.
- Cheriton, D. and 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.
- Christofides, N. (1975), *Graph theory: an algorithmic approach*, Academic Press, New York.
- 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*, North-Holland, Amsterdam, 24-30.
- 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. and Graham, B. (1973), An algorithm for determining the chromatic number of a graph, *SIAM J. on Computing*, 2, 311-318.
- Deo, N. (1974), *Graph theory with applications to engineering and computer science*, Prentice-Hall, Englewood Cliffs.
- 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.
- 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.
- Edmonds, J. (1965), Minimum partition of a matroid into independent subsets, *J. Res. Nat. Bur. Standards Sect. B*, 69, 67-72.
- Edmonds, J. and Karp, R. M. (1972), Theoretical improvements in algorithmic efficiency for network flow problems, *J. of the ACM*, 19, 248-264.
- Euler, L. (1736), Solutio problematis ad geometriam situs pertinentis, *Academiae Petropolitanae*, 8, pp. 128-140.
- Even, S. (1973), *Algorithmic combinatorics*, Mac Millan, New York.
- Even, S. (1979), *Graph algorithms*, Computer Science Press, Potomac.
- Even, S. and Tarjan, R. E. (1975), Network flow and testing graph connectivity, *SIAM J. on Comp.*, 4, 507-518.
- Fáry, I. (1948), On Straight line representation of planar graphs, *Acta Sci. Math. Szeged*, 11, 229-233.
- Floyd, R. W. (1957), Nondeterministic algorithms, *J. of the ACM*, 14, 636-644.
- Ford, L. R. and Fulkerson, D. R. (1956), Maximal flow through a network, *Canadian J. Math.*, 8, 399-404.
- Ford, L. R. and 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. and Fulkerson, D. R. (1962), *Flows in networks*, Princeton University Press, Princeton.
- Fulkerson, D. R. and Gross, O. A. (1965), Incidence matrices and interval graphs, *Pacific J. Math.*, 15, 835-855.
- Furtado, A. L. (1973), *Teoria dos grafos: algoritmos*, Livros Técnicos e Científicos, Rio de Janeiro.
- Furtado, A. L., Santos, C. S., Veloso, P. A. e Azevedo, P. A. (1982), *Estrutura de Dados*, Editora Campus, Rio de Janeiro.
- Galil, Z. (1978), A new algorithm for the maximal flow problem, *Proc. 19th Symp. on the Foundations of Computer Science*, IEEE, 231-245.
- Galil, Z. and Naamad, A. (1979), Network flow and generalized path compression, *Proc. 11th Ann. Symp. on Theory of Computing*, ACM 13-26.
- Gallai, T. (1964), Elementare relationen bezüglich der glieder und trennenden 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. and Johnson, D. S. (1976), The complexity of near-optimal graph coloring, *J. of the ACM*, 23, 43-49.
- Garey, M. R. and Johnson, D. S. (1978), Strong NP-completeness results: motivation, examples and implications, *J. of the ACM*, 25, 439-508.

- Garey, M. R. and Johnson, D. S. (1979), *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco.
- 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. and Hoffman, A. J. (1964), A characterization of comparability graphs and of interval graphs, *Canad. J. of Math.*, 16, 539-548.
- Golomb, S. W. and Baumert, L. D. (1965), Backtrack programming, *J. of the ACM*, 12, 516-524.
- Golumbic, M. C. (1980), *Algorithmic graph theory and perfect graphs*, Academic Press, New York.
- Gondran, M. et Minoux, M. (1979), *Graphes et algorithmes*, Editions Eyrolles, Paris.
- Goodman, S. E. and Hedetniemi, S. T. (1977), *Introduction to design and analysis of algorithms*, McGraw-Hill, New York.
- Greene, D. H. and Knuth, D. E. (1981), *Mathematics for the analysis of algorithms*, Birkhäuser, Boston.
- Hadlock, F. O. (1974), Minimum spanning forests of bounded trees, *Proc. 5th Southeastern Conference on Combinatorics, Graph Theory and Computing*, Utilitas Mathematica Pub., Winnipeg, 449-460.
- Hall, P. (1935), On representations of subsets, *J. London Math. Soc.*, 10, 26-30.
- Harary, F. (1969), *Graph theory*, Addison-Wesley, Reading.
- Harary, F., Norman, R. Z. and Cartwright, D. (1965), *Structural models: an introduction to the theory of directed graphs*, John Wiley, New York.
- Harary, F. and Prins, G. (1967), The block-cutpoint-tree of a graph, *Publ. Math. Debrecen*, 13, 103-107.
- Harrison, M. C. (1973), *Data structures and programming*, Scott, Foresman & Co., Glenview.
- Heawood, P. J. (1890), Map colour theorems, *Quart. J. Math.*, 24, 332-338.
- Hopcroft, J. and Tarjan, R. (1973a), Dividing a graph into tri-connected components, *SIAM J. Comp.*, 2, 135-158.
- Hopcroft, J. and 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, 549-568.
- Hopcroft, J. E. and Ullman, J. D. (1969), *Formal languages and their relation to automata*, Addison-Wesley, Reading.
- Hopcroft, J. E. and Ullman, J. D. (1979), *Introduction to automata theory, languages and computation*, Addison-Wesley, Reading.
- Horowitz, E. and Sahni, S. (1976), *Fundamentals of data structures*, Computer Science Press, Woodland Hills.
- Horowitz, E. and Sahni, S. (1978), *Fundamentals of computer algorithms*, Computer Science Press, Rockville.
- Hu, T. C. (1982), *Combinatorial Algorithms*, Addison-Wesley, Reading.
- Itai, A. and Shiloach, Y. (1979), Maximum flow in planar networks, *SIAM J. on Comp.*, 8, 135-150.
- Johnson, D. B. (1975), Finding all the elementary circuits of a directed graph, *SIAM J. Comp.*, 4, 77-84.
- Johnson, D. S. (1981), The NP-completeness column: an ongoing guide, *Journal of Algorithms*, a partir dez./1981.
- 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. and Thatcher, J. W., eds., *Complexity of computer computations*, Plenum Press, New York, 85-103.
- 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.
- Knuth, D. E. (1968), *The art of computer programming, volume I: Fundamental algorithms*, Addison-Wesley, Reading.
- Knuth, D. E. (1973), *The art of computer programming, volume III: Sorting and searching*, Addison-Wesley, Reading.
- 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.
- 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 travelling 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. and Rinnooy Kan, A. H. G. (1978), Computer aided complexity classification of deterministic scheduling problems, *Mathematisch Centrum*, Amsterdam.
- 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. (1976), *Combinatorial optimization: networks and matroids*, Holt, Rinehart and Winston, New York.
- Levin, L. A. (1973), Universal sorting problems, *Problemy Peredací Informacii*, 9, 115-116.
- Lovász, L. (1979), *Combinatorial problems and exercises*, North-Holland, Amsterdam.
- 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, I., Simon, J. e Kowaltowski, T. (1979), *Aspectos teóricos de computação*, Inst. de Mat. Pura e Aplicada (Projeto Euclides), Rio de Janeiro.
- Lucena, C. J. P. (1972), *Introdução às estruturas de informação*, Livros Técnicos e Científicos, Rio de Janeiro.
- Lueker, G. S. (1974), Structured breadth first search and chordal graphs, *Tech. Rep. TR-158*, Princeton Univ., Princeton.
- 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. Develop.*, 170-180.
- Machtey, M. and Young, P. (1978), *An introduction to the general theory of algorithms*, North-Holland, New York.
- Malhotra, V. M., Pramodh Kumar, M. and 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. and Isaacson, J. D. (1972), Graph coloring algorithms, in Read, R. C., ed., *Graph Theory and Computing*, Academic Press, New York, 109-122.
- 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, Univ. Fed. do Rio de Janeiro, Rio de Janeiro.
- Micali, S. and 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.
- Mycielski, J. (1955), Sur le coloriage des graphs, *Colloq. Math.*, 3, 161-162.
- Nijenhuis, A. and Wilf, H. S. (1975), *Combinatorial algorithms*, Academic Press, New York.
- Oliveira, A. A. F. and 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*, Academic Press, New York.
- Pacitti, T. e Atkinson, C. P. (1975), *Programação e métodos computacionais*, vols. 1 e 2, Livros Técnicos e Científicos, Rio de Janeiro.
- Page, E. S. and Wilson, L. B. (1973), *Information representation and manipulation in a computer*, Cambridge University Press, London.
- Page, E. S. and Wilson, L. B. (1979), *An introduction to computational combinatorics*, Cambridge University Press, Cambridge.
- Papadimitriou, C. H. and Steiglitz, K. (1982), *Combinatorial optimization: algorithms and complexity*, Prentice-Hall, Englewood Cliffs.
- Petersen, J. (1891), Di Theorie der regulären Graphen, *Acta Math.*, 15, 193-220.
- Pfaltz, J. C. (1977), *Computer data structures*, McGraw-Hill, New York.
- 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.
- Rabin, M. O. (1976), *Probabilistic algorithms, algorithms and complexity: new directions and recent results*, J. F. Tramb, ed., Academic Press, New York, 21-40.
- Rabuske, M. A. (1981), *Algumas contribuições à aplicação da teoria dos hipergrafos*, Tese de doutorado, COPPE, Univ. Fed. do Rio de Janeiro, Rio de Janeiro.
- Read, R. C. and 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. and Deo, N. (1977), *Combinatorial algorithms: theory and practice*, Englewood Cliffs, New York.

- Rose, D. J. and Tarjan, R. E. (1975), Algorithmic aspects of vertex elimination, *Proc. 7th. Ann. ACM Symp. Theory Comput.*, 245-254.
- Rose, D. J., Tarjan, R. E. and Lueker, G. S. (1976), Algorithmic aspects of vertex elimination on graphs, *SIAM J. Comp.*, 5, 266-283.
- Saaty, T. and Kainen, P. (1977), *The four color problem: assaults and conquests*, McGraw-Hill, New York.
- Santos, R. N. (1979), *Obtenção do número cromático de um grafo*, Tese de mestrado, COPPE/UFRJ, Rio de Janeiro.
- Savulecu, S. C. (1980), *Grafos, dígrafos e redes elétricas*, Inst. Bras. Ed. Cient., São Paulo.
- Schrader, R. (1981), *A note on approximation algorithms for graph partitioning problems*, Rep. n° 81187-OR, Institut für Ökonometrie und Operations Research, Universität Bonn, Bonn.
- Sethi, R., Complete register allocation problems, *SIAM J. Comp.*, 4, 226-248.
- Shiloach, Y. (1978), *An O(n^{1/2} log² n) maximum flow algorithm*, Tech. Rep. STAN-CS-78-802, Comp. Sc. Dept., Stanford Univ., Stanford.
- Simon, Imre (1981), *Linguagens formais e autômatos*, 2^a Escola de Computação, Inst. de Mat., Estat. e Ciência da Computação, Univ. de Campinas, Campinas.
- Simon, Istvan (1979), *Introdução à teoria de complexidade de algoritmos*, 1^a Escola de Computação, Inst. de Mat. e Estat., Univ. de São Paulo, São Paulo.
- Sleator, D. D. K. (1980), *An O(nmlogn) algorithm for maximum network flow*, Ph. D. thesis, Comp. Sc. Dept., Stanford Univ., Stanford.
- Standish, T. A. (1980), *Data structure techniques*, Addison-Wesley, Reading.
- Szwarcfiter, J. L. (1980), *On a class of acyclic directed graphs*, Mem. UCB/ERL M80/6, Elec. Res. Lab., University of California, Berkeley.
- Szwarcfiter, J. L., Persiano, R. C. M. e Oliveira, A. A. F. (1981), *Um problema em orientação de grafos*, Relatórios Técnicos do Programa de Engenharia de Sistemas e Computação ES 08-81, COPPE/UFRJ, Rio de Janeiro.
- Tarjan, R. (1972), Depth-first search and linear graph algorithms, *SIAM J. Comp.*, 1, 146-160.
- Tarjan, R. (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.
- Tarry, G. (1895), Les problèmes des labyrinthes, *Nouvelles Ann. de Math.*, 14, p. 187.
- Terada, R. (1982), *Desenvolvimento de algoritmos e complexidade de computação*,
- 3^a Escola de Computação, Deptº de Informática, Pontifícia Univ. Católica, Rio de Janeiro.
- Turing, A. (1936), On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.*, Ser. 2, 42, 230-265 e 43, 544-546.
- Tutte, W. T. (1966), *The connectivity of graphs*, Toronto University Press, Toronto.
- Valdes, J. (1978), *Parsing flowcharts and series parallel graphs*, Tech. Rep. STAN-CS-78-682, Comp. Sc. Dept., Stanford Univ., Stanford.
- Valdes, J., Tarjan, R. E. and Lawler, E. L., The recognition of series parallel digraphs, *Proc. 11 th Ann. ACM Symp. on Theory of Comp.*, Atlanta.
- Veloso, P. A. S. (1979), *Máquinas e linguagens: uma introdução à teoria de autômatos*, 1^a Escola de Computação, Inst. de Mat. e Estat., Univ. de São Paulo, São Paulo.
- Wakabayashi, Y. (1977), *Sobre grafos hamiltonianos*, Tese de mestrado, Inst. de Mat. e Estat., Univ. de São Paulo, São Paulo.
- 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*, Pergamon Press, Oxford.
- 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.
- Wilson, R. J. and Beineke, L. W. (1970), *Applications of graph theory*, Academic Press, New York.
- Yao, A. C. C. (1975), An O(|E| log log |V|) algorithm for finding minimum spanning trees, *Info. Proc. Let.*, 4, 21-23.
- Zykov, A. A. (1949), On some properties of linear complexes. *Mat. Sbornik*, 24, 163-188 (Amer. Math. Soc. translation N. 79, 1952).

ÍNDICE DE ALGORITMOS

Inversão dos termos de uma seqüência	23
Ordenação (I)	27
Ordenação (II)	28
Ordenação (III)	28
Determinação de um ciclo euleriano de um grafo	39
Determinação do centro de uma árvore	47
Construção de uma estrutura de adjacências de um dígrafo	76
Ordenação por caixas	76
Coloração aproximada (I)	78
Coloração aproximada (II)	78
Ordenação topológica	80
Busca em grafos — caso geral	88
Busca em profundidade — grafos	89
Determinação das articulações de um grafo	93
Determinação dos componentes biconexos de um grafo	93
Busca em profundidade — dígrafos	95
Determinação dos componentes fortemente conexos de um dígrafo	101
Busca em largura — grafos	103
Busca em largura lexicográfica — grafos	109
Reconhecimento de esquemas de eliminação perfeita	115
Reconhecimento de grafos cordais	116
Busca irrestrita em profundidade — grafos	117
Algoritmo guloso	125
Construção da árvore geradora máxima de um grafo	129
Particionamento de uma árvore em subárvores de peso total máximo	139
Coloração exata	145
Determinação do fluxo máximo em uma rede (I)	156
Determinação do fluxo máximo em uma rede (II)	158
Determinação do fluxo máximo em uma rede (III)	161
Construção de uma rede de camadas	159
Determinação de um fluxo maximal em uma rede de camadas	163

ÍNDICE DE ATRIBUÍDOS

ÍNDICE ALFABÉTICO

A

- Aho, A.V., 34, 197
algoritmo, 20
eficiente, 169
exponencial, 173
guloso, 125-126
ótimo, 26
polinomial, 173
pseudopolinomial, 194-196
alteração estrutural, 142
altura de uma árvore, 50
ancestral, 50
Andrade, M.C.O., 74
Appel, K., 18, 34, 74
aresta, 35
contrária, 153
convergente, 64
de árvore, 89, 95, 104
de avanço, 95
de corte, 132
de cruzamento, 95
de retorno, 89, 95
direta, 153
divergente, 64
implícita por transitividade, 120
irmão, 104
paralela, 37
primo, 104
saturada, 151
tio, 104
articulação, 53, 92-95

B

- Baase, S., 34, 197
Barbosa, R.M., 73
Barron, D.W., 84
Baumert, L.D., 124
Beineke, L.W., 73
Bellman, R.E., 148
Berge, C., 73
Berztiss, A.T., 34
biconectividade, 91-94
Biggs, N.L., 34

bloco, 21, 54-55
especial, 22-23
Boaventura Netto, P.O., 73
Bollobás, B., 73
Bondy, J.A., 73
Busacker, R.G., 73
busca
binária, 28
completa, 99
em largura, 103-105
em largura lexicográfica, 105-110
em profundidade (grafos), 88-91
em profundidade (digráficos), 95-99
em profundidade lexicográfica, 122
geral, 88
irrestrita, 116-118

C

cadeia, 166
caixeiro viajante, 172, 174
caminho, 37
aumentante, 154
de acréscimo de fluxo, 154
elementar, 37
euleriano, 38
hamiltoniano, 38
mínimo condicionado, 196
simples, 37
capacidade
de aresta, 149
de corte, 151
de vértice, 163
Capobianco, M., 73
Carré, B., 73
Cartwright, D., 73
Carvalho, R.L., 34
Cayley, A., 18, 33, 74
centro
de um grafo, 45, 72
de uma árvore, 46-47
Cheriton, D., 148
Cherkassky, B., 166
Christofides, N., 73
ciclo, 38
elementar, 38
euleriano, 38
fundamental, 48
hamiltoniano, 18, 38, 60-61, 72-74,

177, 181
simples, 38
cláusula, 175
clique, 42, 176, 187, 190, 192
máxima, 183
cobertura
de ciclos, 166
de vértices, 176, 182, 190
por cadeias
Cobham, A., 197
coloração, 18-19, 34, 83, 178
aproximada, 77-80
de mapas, 64
mínima, 61, 142-146
complemento
de um grafo, 40
de um problema, 184-186
complexidade
de algoritmos, 24-29
de melhor caso, 26
de pior caso, 26
local, 25
local assintótica, 25
componentes
biconexos, 53
conexos, 38-66
fortemente conexos, 99-102
comprimento de caminho, 37, 157
connectividade
de arestas, 31
de vértices, 52-56, 72, 73
conjunção, 175
conjunto
de arestas de realimentação, 177
de dados, 166
de vértices de realimentação, 178
fundamental de ciclos, 48
incomparável, 165
independente de vértices, 42, 175, 181, 187, 190
maximal, 38
minimal, 38
parcialmente ordenado, 66-67, 80
Co-NP-completo, 195
Cook, S.A., 197
corda, 110
Corneil, D.G., 148
corte, 152
de arestas, 52

de vértices, 52
mínimo, 152, 156

D

demarcador, 93
Deo, N., 34, 73, 197
descendente, 49
destino, 149
diagrama de Hasse, 66-67
dígrafo, 64-69, 72-73
acíclico, 66, 73
desconexo, 66
estruturado em árvore, 121
fortemente conexo, 66
fracamente conexo, 66
série paralelo, 121
unilateralmente conexo, 66

Dijkstra, E.W., 148
Dilworth, R.P., 167
Dinic, E.A., 166
Dirac, G.A., 74, 124
disjunção, 175
distância, 39

E

Edmonds, J., 166, 197
elo, 48
emparelhamento, 166
esquema de eliminação perfeita, 112
estrutura
de adjacências, 70, 76, 83
de dados 29-32
Euler, L., 18, 34, 56
Even, A., 34, 73, 167, 197
excentricidade, 45, 72
expressão booleana, 175
extensão de problema, 192-194
extremidade, 35
extremo, 35

F

face, 56
Fáry, I., 74
fechamento transitivo, 66-68
fila, 31
dupla, 32
filho, 51

floresta, 43
de profundidade, 98
geradora, 47
Floyd, R., 124
fluxo, 149
em redes, 149-167
illegal, 150
maximal, 151
máximo, 151-155
no corte, 151
folha, 43, 50
fonte, 64
Ford, L.R., 166
forma normal conjuntiva, 175
fronde, 89
Fulkerson, D.R., 124, 166
Furtado, A.L., 34, 73

G

Galil, Z., 166
Gallai, T., 74
Garey, M.R., 84, 197
Ghouilla-Houri, A., 74
Gilmore, P.C., 74
Golomb, S.W., 124
Golumbic, M.C., 73, 124
Gondran, M., 73
Gonzaga, C.C., 166
Goodman, S.E., 34
Graham, B., 148
grafo, 35
acíclico, 37
biconexo, 53-55, 60
bipartite, 41-42, 63, 73
bloco articulação, 72, 74
completo, 41
conexo, 38
cordal, 110-118
de comparabilidade, 73-74
de Petersen, 72, 74
desconexo, 66-71, 73-74
euleriano, 38, 40
extremo, 73-74
hamiltoniano, 18, 38, 60-62, 72-74, 177, 181
imersível em uma superfície, 56
isomorfo, 36
k-colorível, 64, 73

k-conexo em arestas, 53
k-conexo em vértices, 53, 73
k-crítico, 63, 72-75
não direcionado, 35
outerplanar, 72-73
perfeito, 73
planar, 56, 64
regular, 37, 73
rotulado, 43
subjacente, 65
triangularizado, 110-118
trivial, 35
totalmente desconexo, 38

grau, 37
de entrada, 64
de saída, 64

Greene, D.H., 34
Gross, O.A., 124
Guthrie, F., 18, 34

H

Haddock, F.O., 148
Haken, W., 18, 34, 74
Hall, P., 167
Harary, F., 73
Harrison, M.C., 34
Heawood, P.J., 34
Hedetniemi, S.T., 34
Hoffman, A.J., 74
Hopcroft, J.E., 34, 124, 197
Horowitz, E., 34, 197
Hu, T.C., 34

I

instância, 170
inversão, 27
de uma seqüência, 23
irmão, 49
Isaacson, J.D., 84
isomorfismo
de grafos, 35-36
de subgrafos, 178, 192-193

Itai, A., 166

J

Johnson, D.B., 124
Johnson, D.S., 84, 197

Jordan, C., 74

K

Kahn, A.B., 84
Kainen, P., 73
Karp, R.M., 166, 197
Karzanov, A.V., 166
Kase, R.H., 84
Kempe, A.B., 34
Kirchhoff, G., 18, 33, 74
Knuth, D.E., 34, 74, 84, 124, 197
König, D., 34
Kowaltowski, T., 34
Kruskal Jr., J.B., 148
Kuratowski, K., 19, 34, 74

L

Lageweg, B.J., 197
largura de um vértice, 104
Lasser, D.J., 84
Lawler, E.L., 34, 124, 148, 197
Lenstra, J.K., 197
Levin, L.A., 197
limite inferior, 26, 83
linguagem de apresentação de algoritmos, 20-23
lista, 29
circular, 30
de adjacências, 71-72, 76
duplamente encadeada, 31
encadeada, 30
seqüencial, 29
vazia, 31
Lloyd, E.K., 34
Lovász, L., 73
Lucas, E., 124
Lucchesi, C.L., 34, 73
Lucena, C.J.P., 34
Lueker, G.S., 124
Lukes, J.A., 148

M

Machtey, M., 34
Maheshwari, S.N., 166
Malhotra, V.M., 166
mapa, 64-65
Marble, G., 84

matriz

de adjacências, 69-70, 73, 84
de incidências, 70
Matula, D.W., 84
Menger, K., 19, 34, 74
Méxas, M.P., 124
Micali, S., 167
Minoux, M., 73
Mitchell, J., 84
Molluzzo, J.C., 73
multigrafo, 37
Murty, U.S.R., 74
Mycielski, J., 74

N

NP, 179, 183
Naamad, A., 166
Nievergelt, J., 197
Nijenhuis, A., 34
nível 48, 159
Norman, R.Z., 73
notação 0, 25
notação Ω , 26
NP-completo, 184
forte, 195-196
fraco, 195-196
NP-difícil, 188
número cromático, 62, 73, 74, 77
números
compostos, 185
primos, 185

O

Oliveira, A.A.F., 124, 166
ordem de nível, 86
ordenação, 27-29
lexicográfica, 107
parcial, 66-67
por caixas, 76-77, 79, 83
topológica, 79-81, 84
Ore, O., 73
origem, 149

P

P, 173-174
Pacitti, T., 34
Page, E.S., 34

pai, 49

Papadimitriou, C.H., 34, 197

particionamento
de árvores, 132-142, 194-196
de grafos, 147
Persiano, R.C.M., 124
Petersen, J., 74
Pfaltz, J.C., 34
pilha, 31
planaridade, 56-60, 73, 74
poço, 64
Pombo, H.C.R., 34
ponte, 53
Ponte de Königsberg, 18
posto, 48

Pramod Kumar, M., 166
Pratt, V., 197
preordem, 86
Prim, R.C., 148
Prins, G., 74
problema
algorítmico, 170
das quatro cores, 18, 34, 64, 73-74
de decisão, 170-173
de localização, 171
de otimização, 171
equivalente, 188
intratável, 169
numérico, 195
tratável, 169
profundidade
de entrada, 90, 96
de saída, 90, 96
programação dinâmica, 130-132

R

Rabuske, M.A., 73
raiz
de um dígrafo, 66
de uma árvore, 50-55
de uma busca, 86
RAM, 24, 197
Read, R.C., 124
recursão, 81
rede, 149
de camadas, 159
residual, 153
Redei, L., 74

redução transitiva, 66-68, 73

Reingold, E.M., 34, 197

representação

de grafos, 68-71, 75

geométrica, 35, 71

restrição de problema, 192-196

Rinnooy Kan, A.H.G., 197

rótulo, 43

Rose, D.J., 124

S

Saaty, T.L., 73

Sahni, S., 34, 197

Santos, G.S., 34

Santos, R.N., 84

satisfabilidade, 174, 181, 189, 192-196

Savulescu, S.C., 73

Schrader, R., 148

separador, 110

seqüência de Fibonacci, 131

Sethi, R., 124

Shiloach, Y., 166

Simon, Imre, 34

Simon, Istvan, 34

Simon, J., 34

sistema de representantes distintos, 166

Sleator, D.D.K., 166

Standish, T.A., 34

Steiglitz, K., 34, 197

subárvore, 50

parcial, 50

subdivisão

de uma aresta, 59

de um grafo, 59

subgrafo, 42

de espalhamento, 47

gerador, 47

induzido, 42

sumidouro, 64

Szwarcfiter, J.L., 124

T

Tarjan, R.E., 34, 124, 148, 167

Tarry, G., 124

Terada, R., 34

torneio, 74, 73,

trajeto, 37

transformação polinomial, 186-188

Trémaux, 124

triângulo, 37, 73, 74

Turing, A., 34

Tutte, W.T., 73

U

Ullman, J.D., 34, 197

V

Valdes, J., 124

valor

de uma expressão booleana, 185

de um fluxo, 150

Vazirani, V.V., 167

Veloso, P.A., 34

vértice, 35

adjacente, 35

alcançável, 37, 66

atingível, 37, 66

convergente, 64

divergente, 64

forte, 100

interior, 43

isolado, 38

saturado, 151

simplicial, 110

vetor 21

W

Wakabayashi, Y., 74

Weide, B., 34

Wells, M.B., 34

Whitney, H., 74

Wilf, H., 34

Wilson, L.B., 34

Wilson, R.J., 73

Y

Yao, A.C.A., 148

Young, P., 34

Zykov, A.A., 148